defcon Documentation

Release 0.7.0

Tal Leming

Aug 20, 2023

Contents

1 Basic Usage 3

2 Concepts 5

3 Objects 9

4 Dependencies 97

5 Optional Dependencies 99

6 Indices and tables 101

Python Module Index 103

Index 105

defcon is a set of UFO based objects optimized for use in font editing applications. The objects are built to be lightweight, fast and flexible. The objects are very bare-bones and they are not meant to be end-all, be-all objects. Rather, they are meant to provide *base functionality* so that you can focus on your application's behavior, not *object observing* or *maintaining cached data*.

CHAPTER 1

Basic Usage

defcon is very easy to use:

```
from defcon import Font
font = Font()
# now do some stuff!
```

CHAPTER 2

Concepts

2.1 Notifications

defcon uses something similar to the Observer Pattern for inter-object communication and object observation. This abstraction allows you to cleanly listen for particular events happening in particular objects. You don't need to wire up lots of hooks into the various objects or establish complex circular relationships thoughout your interface code. Rather, you register to be notified when something happens in an object. In defcon, these are referred to as *notifications*. For example, I want to be notified when the my font changes:

```
class MyInterface(object):

    # random code here, blah, blah.

    def setGlyph(self, glyph):
        glyph.addObserver(self, "glyphChangedCallback", "Glyph.Changed")

    def glyphChangedCallback(self, notification):
        glyph = notification.object
        print("the glyph (%s) changed!" % glyph.name)
```

When the glyph is changed in anyway by anyone, it posts a "Glyph.Changed" notification to all registered observers. My method above is called when this happens and I can react as needed.

The *NotificationCenter* object implements all of this. However, all objects derived from `dercon.BaseObject` have a simplified API for tapping into notifications. Each object posts its own unique notifications, so look at the relevant reference for information about the available notifications.

2.1.1 Don't Forget removeObserver

The only real gotcha in this is that you must remove the observer from the observed object when the observation is no longer needed. If you don't do this and the observed object is changed, it will try to post a notification to the object you have discarded. That could lead to trouble.

2.2 Subclassing

The defcon objects are built to have basic functionality. Your application can, and should, have its own functionality that is not part of the standard defcon repertoire. The objects are built with this in mind – they are built to be subclassed and extended. This is done easily:

```
from defcon import Glyph

class MyCustomGlyph(Glyph):

  def myCustomMethod(self):
    # do something to the glyph data
```

When it is time to load a font, you pass this custom class to the Font object:

```
from defcon import Font

font = Font(glyphClass=MyCustomGlyph)
```

When a glyph is loaded, the glyph class you provided will be used to create the glyph object.

2.3 External Changes

It may be advantageous for your application to notice changes to a UFO that were made outside of your application. the `Font` object can help you with this. This object has a `testForExternalChanges()` method. This method will compare the data that has been loaded into the font, glyphs, etc. with the data in the UFO on disk. It will report anything that is different from when the UFO was last loaded/saved.

To do this in a relatively effiecient way, it stores the modification data and raw text of the UFO file inside the object. When the `testForExternalChanges()` method is called, the modification date of the UFO file and the stored modification date are compared. A mismatch between these two will trigger a comparison between the raw text in the UFO file and the stored raw text. This helps cut down on a significant number of false positives.

The `testForExternalChanges()` method will return a dictionary describing what could have changed. You can then reload the data as appropriate. The `Font` object has a number of *reload* methods specifically for doing this.

2.3.1 Scanning Scheduling

defcon does not automatically search for changes, it is up to the application to determine when the scanning should be performed. The scanning can be an expensive operation, so it is best done at key moments when the user *could* have done something outside of your application. A good way to do this is to catch the event in which your application/document has been selected after being inactive.

2.3.2 Caveats

There are a couple of caveats that you should keep in mind:

1. If the object has been modified and an external change has happened, the *object* is considered to be the most current data. External changes will be ignored. *This may change in the future. I'm still thinking this through.*
2. The font and glyph data is loaded only as needed by defcon. This means that the user could have opened a font in your application, looked at some things but not the "X" glyph, switched out of your application, edited the GLIF file for the "X" glyph and switched back into your application. At this point defcon will not notice that

the “X” has changed because it has not yet been loaded. This probably doesn’t matter as when the “X” is finally loaded the new data will be used. If your application needs to know the exact state of all objects when the font is first created, preload all font and glyph data.

2.4 Representations

One of the painful parts of developing an app that modifies glyphs is managing the visual representation of the glyphs. When the glyph changes, all representations of it in cached data, the user interface, etc. need to change. There are several ways to handle this, but they are all cumbersome. defcon gives you a very simple way of dealing with this: *representations* and *representation factories*.

Note: Representations have been extended to allow other font object classes, so that fonts, layers, glyphs, contours, etc. can have representations too.

2.4.1 Representations and Representation Factories

A *representation* is an object that represents a glyph. As mentioned above, it can be a visual representation of a glyph, such as a NSBezierPath. Representations aren’t just limited to visuals, they can be any type of data that describes a glyph or something about a glyph, for example a string of GLIF text, a tree of point location tuples or anything else you can imagine. A *representation factory* is a function that creates a representation. You don’t manage the representations yourself. Rather, you register the factory and then ask the glyphs for the representations you need. When the glyphs change, the related representations are destroyed and recreated as needed.

2.4.2 Example

As an example, here is a representation factory that creates a NSBezierPath representation:

```
def NSBezierPathFactory(glyph):
    from fontTools.pens.cocoaPen import CocoaPen
    pen = CocoaPen(glyph.getParent())
    glyph.draw(pen)
    return pen.path
```

To register this factory, you do this:

```
from defcon import Glyph, registerRepresentationFactory
registerRepresentationFactory(Glyph, "NSBezierPath", NSBezierPathFactory)
```

Now, when you need a representation, you simply do this:

```
path = glyph.getRepresentation("NSBezierPath")
```

Not only do you only have to register this *once* to be able get the representation for *all* glyphs, the representation is always up to date. So, if you change the outline in the glyph, all you have to do to get the updated representation is:

```
path = glyph.getRepresentation("NSBezierPath")
```

2.4.3 Implementation Details

Representation Factories

Representation factories should be functions that accept a font object class (such as *Glyph*, *Font*, *Contour* etc.) as first argument. After that, you are free to define any keyword arguments you need. You must register the factory with the `registerRepresentationFactory` function. When doing this, you must define a unique name for your representation. The recommendation is that you follow the format of "applicationOrPackageName.representationName" to prevent conflicts. Some examples:

```
registerRepresentationFactory(Glyph, "MetricsMachine.groupEditorGlyphCellImage", ␣
↪groupEditorGlyphCellImageFactory)
registerRepresentationFactory(Glyph, "Prepolator.previewGlyph", previewGlyphFactory)
```

Representations

Once the factory has been registered, glyphs will be able to serve the images. You can get the representation like this:

```
image = glyph.getRepresentation("MetricsMachine.groupEditorGlyphCellImage")
```

You can also pass keyword arguments when you request the representation. For example:

```
image = glyph.getRepresentation("MetricsMachine.groupEditorGlyphCellImage", ␣
↪cellSize=(40, 40))
```

These keyword arguments will be passed along to the representation factory. This makes it possible to have very dynamic factories.

All of this is highly optimized. The representation will be created the first time you request it and then it will be cached within the glyph. The next time you request it, the cached representation will be returned. If the glyph is changed, the representation will automatically be destroyed. When this happens, the representation will not be recreated automatically. It will be recreated the next time you ask for it.

Destroying representations

You can now also specify which notifications should destroy representations. (Previously, any change to a glyph would destroy all representations. That wasn't ideal. Changing the glyph.note would destroy the expensive-to-calculate bounding box representation.)

Each class has a list of notifications that it posts. When you register a factory, you can give a list of notification names that should destroy representations created by the factory you are registering. Here's an example:

```
def layerCMAPRepresentationFactory(layer):
   cmap = {}
   for g in layer:
       if g.unicode is not None:
           cmap[chr(g.unicode)] = g.name
   return cmap

registerRepresentationFactory(Layer, "CMAP", layerCMAPRepresentationFactory, ␣
↪destructiveNotifications=["Layer.GlyphUnicodesChanged"])
```

CHAPTER 3

Objects

3.1 Font

See also:

***Notifications*:** The Font object uses notifications to notify observers of changes.

***External Changes*:** The Font object can observe the files within the UFO for external modifications.

3.1.1 Tasks

File Operations

- *`Font`*
- *`save()`*
- *`path`*
- *`ufoFormatVersion`*
- *`testForExternalChanges()`*
- *`reloadInfo()`*
- *`reloadKerning()`*
- *`reloadGroups()`*
- *`reloadFeatures()`*
- *`reloadLib()`*

Sub-Objects

- *info*
- *kerning*
- *groups*
- *features*
- *layers*
- *lib*
- *unicodeData*

Glyphs

- *Font*
- *newGlyph()*
- *insertGlyph()*
- keys()

Layers

- *newLayer()*

Reference Data

- *glyphsWithOutlines*
- *componentReferences*
- *bounds*
- *controlPointBounds*

Changed State

- *dirty*

Notifications

- *dispatcher*
- *addObserver()*
- *removeObserver()*
- *hasObserver()*

Font

class defcon.**Font**(*path=None*, *kerningClass=None*, *infoClass=None*, *groupsClass=None*, *featuresClass=None*, *libClass=None*, *unicodeDataClass=None*, *layerSetClass=None*, *layerClass=None*, *imageSetClass=None*, *dataSetClass=None*, *guidelineClass=None*, *glyphClass=None*, *glyphContourClass=None*, *glyphPointClass=None*, *glyphComponentClass=None*, *glyphAnchorClass=None*, *glyphImageClass=None*)

If loading from an existing UFO, **path** should be the path to the UFO.

If you subclass one of the sub objects, such as *Glyph*, the class must be registered when the font is created for defcon to know about it. The ***Class** arguments allow for individual ovverrides. If None is provided for an argument, the defcon appropriate class will be used.

This object posts the following notifications:

- Font.Changed
- Font.ReloadedGlyphs
- Font.GlyphOrderChanged
- Font.GuidelinesChanged
- Font.GuidelineWillBeDeleted
- Font.GuidelineWillBeAdded

The Font object has some dict like behavior. For example, to get a glyph:

```
glyph = font["aGlyphName"]
```

To iterate over all glyphs:

```
for glyph in font:
```

To get the number of glyphs:

```
glyphCount = len(font)
```

To find out if a font contains a particular glyph:

```
exists = "aGlyphName" in font
```

To remove a glyph:

```
del font["aGlyphName"]
```

addObserver(*observer*, *methodName*, *notification*, *identifier=None*)

Add an observer to this object's notification dispatcher.

- **observer** An object that can be referenced with weakref.
- **methodName** A string representing the method to be called when the notification is posted.
- **notification** The notification that the observer should be notified of.
- **identifier** None or a string identifying the observation. There is no requirement that the string be unique. A reverse domain naming scheme is recommended, but there are no requirements for the structure of the string.

The method that will be called as a result of the action must accept a single *notification* argument. This will be a `defcon.tools.notifications.Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.addObserver(observer=observer, methodName=methodName,
    notification=notification, observable=anObject, identifier=identifier)
```

appendGuideline(*guideline*)
Append **guideline** to the font. The guideline must be a defcon `Guideline` object or a subclass of that object. An error will be raised if the guideline's identifier conflicts with any of the identifiers within the font.

This will post *Font.GuidelinesChanged* and *Font.Changed* notifications.

bounds
The bounds of all glyphs in the font's main layer. This can be an expensive operation.

canRedo()
Returns a boolean indicating whether the undo manager is able to perform a redo.

canUndo()
Returns a boolean indicating whether the undo manager is able to perform an undo.

clearGuidelines()
Clear all guidelines from the font.

This posts a *Font.Changed* notification.

componentReferences
A dict of describing the component relationships in the font's main layer. The dictionary is of form `{base glyph : [references]}`.

controlPointBounds
The control bounds of all glyphs in the font's main layer. This only measures the point positions, it does not measure curves. So, curves without points at the extrema will not be properly measured. This is an expensive operation.

data
The font's `DataSet` object.

destroyAllRepresentations(*notification=None*)
Destroy all representations.

destroyRepresentation(*name*, ***kwargs*)
Destroy the stored representation for **name** and ****kwargs**. If no **kwargs** are given, any representation with **name** will be destroyed regardless of the **kwargs** passed when the representation was created.

dirty
The dirty state of the object. True if the object has been changed. False if not. Setting this to True will cause the base changed notification to be posted. The object will automatically maintain this attribute and update it as you change the object.

disableNotifications(*notification=None*, *observer=None*)
Disable this object's notifications until told to resume them.

- **notification** The specific notification to disable. This is optional. If no *notification* is given, all notifications will be disabled.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.disableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

dispatcher
The `defcon.tools.notifications.NotificationCenter` assigned to this font.

enableNotifications(*notification=None*, *observer=None*)
Enable this object's notifications.

- **notification** The specific notification to enable. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.enableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

features
The font's `Features` object.

findObservations(*observer=None*, *notification=None*, *observable=None*, *identifier=None*)
Find observations of this object matching the given arguments based on the values that were passed during addObserver. A value of None for any of these indicates that all should be considered to match the value. In the case of identifier, strings will be matched using fnmatch.fnmatchcase. The returned value will be a list of dictionaries with this format:

[

{ observer=<…> observable=<…> methodName="…" notification="…" identifier="…"

}

]

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.findObservations(
    observer=observer, observable=anObject,
    notification=notification, identifier=identifier
)
```

getDataForSerialization(***kwargs*)
Return a dict of data that can be pickled.

getRepresentation(*name*, ***kwargs*)
Get a representation. **name** must be a registered representation name. ****kwargs** will be passed to the appropriate representation factory.

getSaveProgressBarTickCount(*formatVersion=None*)
Get the number of ticks that will be used by a progress bar in the save method. Subclasses may override this method to implement custom saving behavior.

glyphOrder
The font's glyph order. When setting the value must be a list of glyph names. There is no requirement, nor guarantee, that the list will contain only names of glyphs in the font. Setting this posts *Font.GlyphOrderChanged* and *Font.Changed* notifications.

glyphsWithOutlines
A list of glyphs containing outlines in the font's main layer.

groups
The font's *Groups* object.

guidelineIndex(*guideline*)
Get the index for **guideline**.

guidelines
An ordered list of `Guideline` objects stored in the font. Setting this will post a *Font.Changed* notification along with any notifications posted by the *Font.appendGuideline()* and *Font.clearGuidelines()* methods.

hasCachedRepresentation(*name*, ***kwargs*)
Returns a boolean indicating if a representation for **name** and ****kwargs** is cached in the object.

hasObserver(*observer*, *notification*)
Returns a boolean indicating is the **observer** is registered for **notification**.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.hasObserver(observer=observer,
    notification=notification, observable=anObject)
```

holdNotifications(*notification=None*, *note=None*)
Hold this object's notifications until told to release them.

- **notification** The specific notification to hold. This is optional. If no *notification* is given, all notifications will be held.
- **note** An arbitrary string containing information about why the hold has been requested, the requester, etc. This is used for reference only.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.holdNotifications(
    observable=anObject, notification=notification, note=note)
```

identifiers
Set of identifiers for the info. This is primarily for internal use.

images
The font's `ImageSet` object.

info
The font's *Info* object.

insertGlyph(*glyph*, *name=None*)
Insert **glyph** into the font's main layer. Optionally, the glyph can be renamed at the same time by providing **name**. If a glyph with the glyph name, or the name provided as **name**, already exists, the existing glyph will be replaced with the new glyph.

insertGuideline(*index*, *guideline*)
Insert **guideline** into the font at index. The guideline must be a defcon `Guideline` object or a subclass of that object. An error will be raised if the guideline's identifier conflicts with any of the identifiers within the font.

This will post *Font.GuidelinesChanged* and *Font.Changed* notifications.

kerning
The font's *Kerning* object.

kerningGroupConversionRenameMaps
The kerning group rename map that will be used when writing UFO 1 and UFO 2. This follows the format defined in UFOReader. This will only not be None if it has been set or this object was loaded from a UFO 1 or UFO 2 file.

layers
The font's `LayerSet` object.

lib
The font's `Lib` object.

newGlyph(*name*)
Create a new glyph with **name** in the font's main layer. If a glyph with that name already exists, the existing glyph will be replaced with the new glyph.

newLayer(*name*)
Create a new `Layer` and add it to the top of the layer order.

This posts *LayerSet.LayerAdded* and *LayerSet.Changed* notifications.

path
The location of the file on disk. Setting the path should only be done when the user has moved the file in the OS interface. Setting the path is not the same as a save operation.

postNotification(*notification*, *data=None*)
Post a **notification** through this object's notification dispatcher.

- **notification** The name of the notification.
- **data** Arbitrary data that will be stored in the `Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.postNotification(
    notification=notification, observable=anObject, data=data)
```

redo()
Perform a redo if possible, or return. If redo is performed, this will post *BaseObject.BeginRedo* and *BaseObject.EndRedo* notifications.

releaseHeldNotifications(*notification=None*)
Release this object's held notifications.

- **notification** The specific notification to hold. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.releaseHeldNotifications(
    observable=anObject, notification=notification)
```

reloadData(*fileNames*)
Reload the data files listed in **fileNames** from the appropriate files within the UFO. When all of the loading is complete, a *Font.ReloadedData* notification will be posted.

reloadFeatures()
Reload the data in the `Features` object from the features.fea file in the UFO.

reloadGlyphs(*glyphNames*)
Deprecated! Use reloadLayers!

Reload the glyphs listed in **glyphNames** from the appropriate files within the UFO. When all of the loading is complete, a *Font.ReloadedGlyphs* notification will be posted.

reloadGroups()
Reload the data in the `Groups` object from the groups.plist file in the UFO.

reloadImages(*fileNames*)
Reload the images listed in **fileNames** from the appropriate files within the UFO. When all of the loading is complete, a *Font.ReloadedImages* notification will be posted.

reloadInfo()
Reload the data in the `Info` object from the fontinfo.plist file in the UFO.

reloadKerning()
Reload the data in the `Kerning` object from the kerning.plist file in the UFO.

This validates the kerning against the groups loaded into the font. If groups are being reloaded in the same pass, the groups should always be reloaded before reloading the kerning.

reloadLayers(*layerData*)
Reload the data in the layers specfied in **layerData**. When all of the loading is complete, *Font.ReloadedLayers* and *Font.ReloadedGlyphs* notifications will be posted. The **layerData** must be a dictionary following this format:

```
{
    "order"   : bool, # True if you want the order releaded
    "default" : bool, # True if you want the default layer reset
    "layers"  : {
        "layer name" : {
            "glyphNames" : ["glyph name 1", "glyph name 2"], # list of glyph␣
↪names you want to reload
            "info"       : bool, # True if you want the layer info reloaded
        }
    }
}
```

reloadLib()
Reload the data in the `Lib` object from the lib.plist file in the UFO.

removeGuideline(*guideline*)
Remove **guideline** from the font.

This will post *Font.GuidelineWillBeDeleted*, *Font.GuidelinesChanged* and *Font.Changed* notifications.

removeObserver(*observer*, *notification*)
Remove an observer from this object's notification dispatcher.

- **observer** A registered object.
- **notification** The notification that the observer was registered to be notified of.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.removeObserver(observer=observer,
    notification=notification, observable=anObject)
```

representationKeys()
Get a list of all representation keys that are currently cached.

save (*path=None*, *formatVersion=None*, *removeUnreferencedImages=False*, *progressBar=None*, *structure=None*)
Save the font to **path**. If path is None, the path from the last save or when the font was first opened will be used.

The UFO will be saved using the format found at `ufoFormatVersion`. This value is either the format version from the exising UFO or the format version specified in a previous save. If neither of these is available, the UFO will be written as format version 3. If you wish to specifiy the format version for saving, pass the desired number as the **formatVersion** argument.

Optionally, the UFO can be purged of unreferenced images during this operation. To do this, pass `True` as the value for the removeUnreferencedImages argument.

'structure' can be either None, "zip" or "package". If it's None, the destination UFO will use the same structure as original, provided that is compatible with any previous UFO at the output path. If 'structure' is "zip" the UFO will be saved as compressed archive, else it is saved as a regular folder or "package".

saveData (*writer*, *saveAs=False*, *progressBar=None*)
Save data. This method should not be called externally. Subclasses may override this method to implement custom saving behavior.

saveFeatures (*writer*)
Save features. This method should not be called externally. Subclasses may override this method to implement custom saving behavior.

saveGroups (*writer*)
Save groups. This method should not be called externally. Subclasses may override this method to implement custom saving behavior.

saveImages (*writer*, *removeUnreferencedImages=False*, *saveAs=False*, *progressBar=None*)
Save images. This method should not be called externally. Subclasses may override this method to implement custom saving behavior.

saveInfo (*writer*)
Save info. This method should not be called externally. Subclasses may override this method to implement custom saving behavior.

saveKerning (*writer*)
Save kerning. This method should not be called externally. Subclasses may override this method to implement custom saving behavior.

saveLib (*writer*, *saveAs=False*, *progressBar=None*)
Save lib. This method should not be called externally. Subclasses may override this method to implement custom saving behavior.

setDataFromSerialization (*data*)
Restore state from the provided data-dict.

tempLib
The font's *tempLib* object.

testForExternalChanges ()
Test the UFO for changes that occured outside of this font's tree of objects. This returns a dictionary describing the changes:

```
{
    "info"     : bool, # True if changed, False if not changed
    "kerning"  : bool, # True if changed, False if not changed
    "groups"   : bool, # True if changed, False if not changed
    "features" : bool, # True if changed, False if not changed
    "lib"      : bool, # True if changed, False if not changed
```

(continues on next page)

(continued from previous page)

```
    "layers"   : {
        "defaultLayer" : bool, # True if changed, False if not changed
        "order"        : bool, # True if changed, False if not changed
        "added"        : ["layer name 1", "layer name 2"],
        "deleted"      : ["layer name 1", "layer name 2"],
        "modified"     : {
            "info"     : bool, # True if changed, False if not changed
            "modified" : ["glyph name 1", "glyph name 2"],
            "added"    : ["glyph name 1", "glyph name 2"],
            "deleted"  : ["glyph name 1", "glyph name 2"]
        }
    },
    "images"   : {
        "modified" : ["image name 1", "image name 2"],
        "added"    : ["image name 1", "image name 2"],
        "deleted"  : ["image name 1", "image name 2"],
    },
    "data"     : {
        "modified" : ["file name 1", "file name 2"],
        "added"    : ["file name 1", "file name 2"],
        "deleted"  : ["file name 1", "file name 2"],
    }
}
```

It is important to keep in mind that the user could have created conflicting data outside of the font's tree of objects. For example, say the user has set `font.info.unitsPerEm = 1000` inside of the font's *Info* object and the user has not saved this change. In the the font's fontinfo.plist file, the user sets the unitsPerEm value to 2000. Which value is current? Which value is right? defcon leaves this decision up to you.

ufoFileStructure
The UFO file structure that will be used when saving. This is taken from a loaded UFO during __init__. If this font was not loaded from a UFO, this will return None until the font has been saved.

ufoFormatVersion
The UFO format major version that will be used when saving. This is taken from a loaded UFO during __init__. If this font was not loaded from a UFO, this will return None until the font has been saved. Deprecated, use ufoFormatVersionTuple instead.

ufoFormatVersionTuple
The UFO format (major, minor) version tuple that will be used when saving. This is taken from a loaded UFO during __init__. If this font was not loaded from a UFO, this will return None until the font has been saved.

undo()
Perform an undo if possible, or return. If undo is performed, this will post *BaseObject.BeginUndo* and *BaseObject.EndUndo* notifications.

undoManager
The undo manager assigned to this object.

unicodeData
The font's *UnicodeData* object.

updateGlyphOrder(*addedGlyph=None*, *removedGlyph=None*)
This method tries to keep the glyph order in sync. This should not be called externally. It may be overriden by subclasses as needed.

3.2 Layer

3.2.1 Layer

class defcon.**Layer**(*layerSet=None*, *glyphSet=None*, *libClass=None*, *unicodeDataClass=None*, *guidelineClass=None*, *glyphClass=None*, *glyphContourClass=None*, *glyphPointClass=None*, *glyphComponentClass=None*, *glyphAnchorClass=None*, *glyphImageClass=None*)

This object represents a layer in a `LayerSet`.

This object posts the following notifications:

- Layer.Changed
- Layer.GlyphsChanged
- Layer.GlyphChanged
- Layer.GlyphWillBeAdded
- Layer.GlyphAdded
- Layer.GlyphWillBeDeleted
- Layer.GlyphDeleted
- Layer.GlyphNameChanged
- Layer.GlyphUnicodesChanged
- Layer.NameChanged
- Layer.ColorChanged
- Layer.LibChanged

The Layer object has some dict like behavior. For example, to get a glyph:

```
glyph = layer["aGlyphName"]
```

To iterate over all glyphs:

```
for glyph in layer:
```

To get the number of glyphs:

```
glyphCount = len(layer)
```

To find out if a font contains a particular glyph:

```
exists = "aGlyphName" in layer
```

To remove a glyph:

```
del layer["aGlyphName"]
```

addObserver(*observer*, *methodName*, *notification*, *identifier=None*)

Add an observer to this object's notification dispatcher.

- **observer** An object that can be referenced with weakref.
- **methodName** A string representing the method to be called when the notification is posted.

- **notification** The notification that the observer should be notified of.
- **identifier** None or a string identifying the observation. There is no requirement that the string be unique. A reverse domain naming scheme is recommended, but there are no requirements for the structure of the string.

The method that will be called as a result of the action must accept a single *notification* argument. This will be a `defcon.tools.notifications.Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.addObserver(observer=observer, methodName=methodName,
    notification=notification, observable=anObject, identifier=identifier)
```

bounds
The bounds of all glyphs in the layer. This can be an expensive operation.

canRedo()
Returns a boolean indicating whether the undo manager is able to perform a redo.

canUndo()
Returns a boolean indicating whether the undo manager is able to perform an undo.

color
The layer's `Color` object. When setting, the value can be a UFO color string, a sequence of (r, g, b, a) or a `Color` object. Setting this posts *Layer.ColorChanged* and *Layer.Changed* notifications.

componentReferences
A dict of describing the component relationships in the layer. The dictionary is of form `{base glyph : [references]}`.

controlPointBounds
The control bounds of all glyphs in the layer. This only measures the point positions, it does not measure curves. So, curves without points at the extrema will not be properly measured. This is an expensive operation.

destroyAllRepresentations(*notification=None*)
Destroy all representations.

destroyRepresentation(*name*, ***kwargs*)
Destroy the stored representation for **name** and ****kwargs**. If no **kwargs** are given, any representation with **name** will be destroyed regardless of the **kwargs** passed when the representation was created.

dirty
The dirty state of the object. True if the object has been changed. False if not. Setting this to True will cause the base changed notification to be posted. The object will automatically maintain this attribute and update it as you change the object.

disableNotifications(*notification=None*, *observer=None*)
Disable this object's notifications until told to resume them.

- **notification** The specific notification to disable. This is optional. If no *notification* is given, all notifications will be disabled.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.disableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

dispatcher
The *defcon.tools.notifications.NotificationCenter* assigned to the parent of this object.

enableNotifications(*notification=None*, *observer=None*)
Enable this object's notifications.

- **notification** The specific notification to enable. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.enableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

findObservations(*observer=None*, *notification=None*, *observable=None*, *identifier=None*)
Find observations of this object matching the given arguments based on the values that were passed during addObserver. A value of None for any of these indicates that all should be considered to match the value. In the case of identifier, strings will be matched using fnmatch.fnmatchcase. The returned value will be a list of dictionaries with this format:

[

{ observer=<…> observable=<…> methodName="…" notification="…" identifier="…"

}

]

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.findObservations(
    observer=observer, observable=anObject,
    notification=notification, identifier=identifier
)
```

font
The *Font* that this layer belongs to.

getDataForSerialization(***kwargs*)
Return a dict of data that can be pickled.

getRepresentation(*name*, ***kwargs*)
Get a representation. **name** must be a registered representation name. ****kwargs** will be passed to the appropriate representation factory.

getSaveProgressBarTickCount(*formatVersion*)
Get the number of ticks that will be used by a progress bar in the save method. This method should not be called externally. Subclasses may override this method to implement custom saving behavior.

glyphsWithOutlines
A list of glyphs containing outlines.

hasCachedRepresentation(*name*, ***kwargs*)
Returns a boolean indicating if a representation for **name** and ****kwargs** is cached in the object.

hasObserver(*observer*, *notification*)
Returns a boolean indicating is the **observer** is registered for **notification**.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.hasObserver(observer=observer,
    notification=notification, observable=anObject)
```

holdNotifications(*notification=None*, *note=None*)
Hold this object's notifications until told to release them.

- **notification** The specific notification to hold. This is optional. If no *notification* is given, all notifications will be held.
- **note** An arbitrary string containing information about why the hold has been requested, the requester, etc. This is used for reference only.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.holdNotifications(
    observable=anObject, notification=notification, note=note)
```

imageReferences
A dict of describing the image file references in the layer. The dictionary is of form `{image file name : [references]}`.

insertGlyph(*glyph*, *name=None*)
Insert **glyph** into the layer. Optionally, the glyph can be renamed at the same time by providing **name**. If a glyph with the glyph name, or the name provided as **name**, already exists, the existing glyph will be replaced with the new glyph.

This posts *Layer.GlyphWillBeAdded*, *Layer.GlyphAdded* and *Layer.Changed* notifications.

keys()
The names of all glyphs in the layer.

layerSet
The *LayerSet* that this layer belongs to.

lib
The layer's *Lib* object.

loadGlyph(*name*)
Load a glyph from the glyph set. This should not be called externally, but subclasses may override it for custom behavior.

name
The name of the layer. Setting this posts *Layer.NameChanged* and *Layer.Changed* notifications.

newGlyph(*name*)
Create a new glyph with **name**. If a glyph with that name already exists, the existing glyph will be replaced with the new glyph.

This posts *Layer.GlyphWillBeAdded*, *Layer.GlyphAdded* and *Layer.Changed* notifications.

postNotification(*notification*, *data=None*)
Post a **notification** through this object's notification dispatcher.

- **notification** The name of the notification.
- **data** Arbitrary data that will be stored in the `Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.postNotification(
    notification=notification, observable=anObject, data=data)
```

redo()
Perform a redo if possible, or return. If redo is performed, this will post *BaseObject.BeginRedo* and *BaseObject.EndRedo* notifications.

releaseHeldNotifications(*notification=None*)
Release this object's held notifications.

- **notification** The specific notification to hold. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.releaseHeldNotifications(
    observable=anObject, notification=notification)
```

reloadGlyphs(*glyphNames*)
Reload the glyphs. This should not be called externally.

removeObserver(*observer*, *notification*)
Remove an observer from this object's notification dispatcher.

- **observer** A registered object.
- **notification** The notification that the observer was registered to be notified of.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.removeObserver(observer=observer,
    notification=notification, observable=anObject)
```

representationKeys()
Get a list of all representation keys that are currently cached.

save(*glyphSet*, *saveAs=False*, *progressBar=None*)
Save the layer. This method should not be called externally. Subclasses may override this method to implement custom saving behavior.

saveGlyph(*glyph*, *glyphSet*, *saveAs=False*)
Save a glyph. This method should not be called externally. Subclasses may override this method to implement custom saving behavior.

setDataFromSerialization(*data*)
Restore state from the provided data-dict.

tempLib
The layer's `tempLib` object.

testForExternalChanges(*reader*)
Test for external changes. This should not be called externally.

undo()
Perform an undo if possible, or return. If undo is performed, this will post *BaseObject.BeginUndo* and *BaseObject.EndUndo* notifications.

undoManager
The undo manager assigned to this object.

unicodeData
The layer's *UnicodeData* object.

3.3 Glyph

See also:

Notifications: The Glyph object uses notifications to notify observers of changes.

Representations: The Glyph object can maintain representations of various arbitrary types.

3.3.1 Tasks

Name and Unicodes

- *name*
- *unicodes*
- *unicode*

Metrics

- *leftMargin*
- *rightMargin*
- *width*

Reference Data

- *area*
- *bounds*
- *controlPointBounds*

General Editing

- *clear()*
- *move()*

Contours

- *Glyph*
- *clearContours()*
- *appendContour()*
- *insertContour()*
- *contourIndex()*

- `autoContourDirection()`
- *correctContourDirection()*

Components

- *components*
- *clearComponents()*
- *appendComponent()*
- *componentIndex()*
- *insertComponent()*

Anchors

- *anchors*
- *clearAnchors()*
- *appendAnchor()*
- *anchorIndex()*
- *insertAnchor()*

Hit Testing

- *pointInside()*

Pens and Drawing

- *getPen()*
- *getPointPen()*
- *draw()*
- *drawPoints()*

Representations

- *getRepresentation()*
- *hasCachedRepresentation()*
- *representationKeys()*
- *destroyRepresentation()*
- *destroyAllRepresentations()*

Changed State

- *dirty*

Notifications

- *dispatcher*
- *addObserver()*
- *removeObserver()*
- *hasObserver()*

Parent

- `getParent()`
- `setParent()`

Glyph

class defcon.**Glyph**(*layer=None*, *contourClass=None*, *pointClass=None*, *componentClass=None*, *anchorClass=None*, *guidelineClass=None*, *libClass=None*, *imageClass=None*)

This object represents a glyph and it contains contour, component, anchor and other assorted bits data about the glyph.

This object posts the following notifications:

- Glyph.Changed
- Glyph.BeginUndo
- Glyph.EndUndo
- Glyph.BeginRedo
- Glyph.EndRedo
- Glyph.NameWillChange
- Glyph.NameChanged
- Glyph.UnicodesChanged
- Glyph.WidthChanged
- Glyph.HeightChanged
- Glyph.LeftMarginWillChange
- Glyph.LeftMarginDidChange
- Glyph.RightMarginWillChange
- Glyph.RightMarginDidChange
- Glyph.TopMarginWillChange
- Glyph.TopMarginDidChange
- Glyph.BottomMarginWillChange
- Glyph.BottomMarginDidChange
- Glyph.NoteChanged
- Glyph.LibChanged

- Glyph.ImageChanged
- Glyph.ImageWillBeCleared
- Glyph.ImageCleared
- Glyph.ContourWillBeAdded
- Glyph.ContourWillBeDeleted
- Glyph.ContoursChanged
- Glyph.ComponentWillBeAdded
- Glyph.ComponentWillBeDeleted
- Glyph.ComponentsChanged
- Glyph.AnchorWillBeAdded
- Glyph.AnchorWillBeDeleted
- Glyph.AnchorsChanged
- Glyph.GuidelineWillBeAdded
- Glyph.GuidelineWillBeDeleted
- Glyph.GuidelinesChanged
- Glyph.MarkColorChanged
- Glyph.VerticalOriginChanged

The Glyph object has list like behavior. This behavior allows you to interact with contour data directly. For example, to get a particular contour:

```
contour = glyph[0]
```

To iterate over all contours:

```
for contour in glyph:
```

To get the number of contours:

```
contourCount = len(glyph)
```

To interact with components or anchors in a similar way, use the `components` and `anchors` attributes.

addObserver(*observer*, *methodName*, *notification*, *identifier=None*)
Add an observer to this object's notification dispatcher.

- **observer** An object that can be referenced with weakref.
- **methodName** A string representing the method to be called when the notification is posted.
- **notification** The notification that the observer should be notified of.
- **identifier** None or a string identifying the observation. There is no requirement that the string be unique. A reverse domain naming scheme is recommended, but there are no requirements for the structure of the string.

The method that will be called as a result of the action must accept a single *notification* argument. This will be a `defcon.tools.notifications.Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.addObserver(observer=observer, methodName=methodName,
    notification=notification, observable=anObject, identifier=identifier)
```

anchorClass
The class used for anchors.

anchorIndex(*anchor*)
Get the index for **anchor**.

anchors
An ordered list of `Anchor` objects stored in the glyph.

appendAnchor(*anchor*)
Append **anchor** to the glyph. The anchor must be a defcon `Anchor` object or a subclass of that object. An error will be raised if the anchor's identifier conflicts with any of the identifiers within the glyph.

This will post a *Glyph.Changed* notification.

appendComponent(*component*)
Append **component** to the glyph. The component must be a defcon `Component` object or a subclass of that object. An error will be raised if the component's identifier conflicts with any of the identifiers within the glyph.

This will post a *Glyph.Changed* notification.

appendContour(*contour*)
Append **contour** to the glyph. The contour must be a defcon `Contour` object or a subclass of that object. An error will be raised if the contour's identifier or a point identifier conflicts with any of the identifiers within the glyph.

This will post a *Glyph.Changed* notification.

appendGuideline(*guideline*)
Append **guideline** to the glyph. The guideline must be a defcon `Guideline` object or a subclass of that object. An error will be raised if the guideline's identifier conflicts with any of the identifiers within the glyph.

This will post a *Glyph.Changed* notification.

area
The area of the glyph's outline.

bottomMargin
The bottom margin of the glyph. Setting this posts *Glyph.HeightChanged*, *Glyph.BottomMarginWillChange*, *Glyph.BottomMarginDidChange* and *Glyph.Changed* notifications among others.

bounds
The bounds of the glyph's outline expressed as a tuple of form (xMin, yMin, xMax, yMax).

canRedo()
Returns a boolean indicating whether the undo manager is able to perform a redo.

canUndo()
Returns a boolean indicating whether the undo manager is able to perform an undo.

clear()
Clear all contours, components, anchors and guidelines from the glyph.

This posts a *Glyph.Changed* notification.

clearAnchors()
Clear all anchors from the glyph.

This posts a *Glyph.Changed* notification.

clearComponents()
Clear all components from the glyph.

This posts a *Glyph.Changed* notification.

clearContours()
Clear all contours from the glyph.

This posts a *Glyph.Changed* notification.

clearGuidelines()
Clear all guidelines from the glyph.

This posts a *Glyph.Changed* notification.

componentClass
The class used for components.

componentIndex(*component*)
Get the index for **component**.

components
An ordered list of `Component` objects stored in the glyph.

contourClass
The class used for contours.

contourIndex(*contour*)
Get the index for **contour**.

controlPointBounds
The control bounds of all points in the glyph. This only measures the point positions, it does not measure curves. So, curves without points at the extrema will not be properly measured.

copyDataFromGlyph(*glyph*)
Copy data from **glyph**. This copies the following data:

width height unicodes note image contours components anchors guidelines lib ==========

The name attribute is purposefully omitted.

correctContourDirection(*trueType=False*, *segmentLength=10*)
Correct the direction of all contours in the glyph.

This posts a *Glyph.Changed* notification.

decomposeAllComponents()
Decompose all components in this glyph. This will preserve the identifiers in the incoming contours and points unless there is a conflict. In that case, the conflicting incoming identifier will be discarded.

This posts *Glyph.ComponentsChanged*, *Glyph.ContoursChanged* and *Glyph.Changed* notifications.

decomposeComponent(*component*)
Decompose **component**. This will preserve the identifiers in the incoming contours and points unless there is a conflict. In that case, the conflicting incoming identifier will be discarded.

This posts *Glyph.ComponentsChanged*, *Glyph.ContoursChanged* and *Glyph.Changed* notifications.

destroyAllRepresentations(*notification=None*)
Destroy all representations.

destroyRepresentation(*name*, ***kwargs*)
Destroy the stored representation for **name** and ****kwargs**. If no **kwargs** are given, any representation with **name** will be destroyed regardless of the **kwargs** passed when the representation was created.

dirty
The dirty state of the object. True if the object has been changed. False if not. Setting this to True will cause the base changed notification to be posted. The object will automatically maintain this attribute and update it as you change the object.

disableNotifications(*notification=None*, *observer=None*)
Disable this object's notifications until told to resume them.

- **notification** The specific notification to disable. This is optional. If no *notification* is given, all notifications will be disabled.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.disableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

dispatcher
The *defcon.tools.notifications.NotificationCenter* assigned to the parent of this object.

draw(*pen*)
Draw the glyph with **pen**.

drawPoints(*pointPen*)
Draw the glyph with **pointPen**.

enableNotifications(*notification=None*, *observer=None*)
Enable this object's notifications.

- **notification** The specific notification to enable. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.enableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

findObservations(*observer=None*, *notification=None*, *observable=None*, *identifier=None*)
Find observations of this object matching the given arguments based on the values that were passed during addObserver. A value of None for any of these indicates that all should be considered to match the value. In the case of identifier, strings will be matched using fnmatch.fnmatchcase. The returned value will be a list of dictionaries with this format:

[

{ observer=<…> observable=<…> methodName="…" notification="…" identifier="…"

}

]

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.findObservations(
    observer=observer, observable=anObject,
```

(continues on next page)

(continued from previous page)

```
    notification=notification, identifier=identifier
)
```

font
The *Font* that this glyph belongs to.

getDataForSerialization(***kwargs*)
Return a dict of data that can be pickled.

getPen()
Get the pen used to draw into this glyph.

getPointPen()
Get the point pen used to draw into this glyph.

getRepresentation(*name*, ***kwargs*)
Get a representation. **name** must be a registered representation name. ****kwargs** will be passed to the appropriate representation factory.

guidelineClass
The class used for guidelines.

guidelineIndex(*guideline*)
Get the index for **guideline**.

guidelines
An ordered list of `Guideline` objects stored in the glyph. Setting this will post a *Glyph.Changed* notification along with any notifications posted by the *Glyph.appendGuideline()* and *Glyph.clearGuidelines()* methods.

hasCachedRepresentation(*name*, ***kwargs*)
Returns a boolean indicating if a representation for **name** and ****kwargs** is cached in the object.

hasObserver(*observer*, *notification*)
Returns a boolean indicating is the **observer** is registered for **notification**.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.hasObserver(observer=observer,
    notification=notification, observable=anObject)
```

height
The height of the glyph. Setting this posts *Glyph.HeightChanged* and *Glyph.Changed* notifications.

holdNotifications(*notification=None*, *note=None*)
Hold this object's notifications until told to release them.

- **notification** The specific notification to hold. This is optional. If no *notification* is given, all notifications will be held.
- **note** An arbitrary string containing information about why the hold has been requested, the requester, etc. This is used for reference only.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.holdNotifications(
    observable=anObject, notification=notification, note=note)
```

identifiers
Set of identifiers for the glyph. This is primarily for internal use.

image
The glyph's `Image` object. Setting this posts *Glyph.ImageChanged* and *Glyph.Changed* notifications.

imageClass
The class used for the image.

insertAnchor(*index*, *anchor*)
Insert **anchor** into the glyph at index. The anchor must be a defcon `Anchor` object or a subclass of that object. An error will be raised if the anchor's identifier conflicts with any of the identifiers within the glyph.

This will post a *Glyph.Changed* notification.

insertComponent(*index*, *component*)
Insert **component** into the glyph at index. The component must be a defcon `Component` object or a subclass of that object. An error will be raised if the component's identifier conflicts with any of the identifiers within the glyph.

This will post a *Glyph.Changed* notification.

insertContour(*index*, *contour*)
Insert **contour** into the glyph at index. The contour must be a defcon `Contour` object or a subclass of that object. An error will be raised if the contour's identifier or a point identifier conflicts with any of the identifiers within the glyph.

This will post a *Glyph.Changed* notification.

insertGuideline(*index*, *guideline*)
Insert **guideline** into the glyph at index. The guideline must be a defcon `Guideline` object or a subclass of that object. An error will be raised if the guideline's identifier conflicts with any of the identifiers within the glyph.

This will post a *Glyph.Changed* notification.

layer
The `Layer` that this glyph belongs to.

layerSet
The `LayerSet` that this glyph belongs to.

leftMargin
The left margin of the glyph. Setting this posts *Glyph.WidthChanged*, *Glyph.LeftMarginWillChange*, *Glyph.LeftMarginDidChange* and *Glyph.Changed* notifications among others.

lib
The glyph's `Lib` object. Setting this will clear any existing lib data and post a *Glyph.Changed* notification if data was replaced.

libClass
The class used for the lib.

markColor
The glyph's mark color. When setting, the value can be a UFO color string, a sequence of (r, g, b, a) or a `Color` object. Setting this posts *Glyph.MarkColorChanged* and *Glyph.Changed* notifications.

move(*values*)
Move all contours, components and anchors in the glyph by **(x, y)**.

This posts a *Glyph.Changed* notification.

name
The name of the glyph. Setting this posts *GLyph.NameChanged* and *Glyph.NameChanged* notifications.

note
An arbitrary note for the glyph. Setting this will post a *Glyph.Changed* notification.

pointClass
The class used for points.

pointInside(*coordinates*, *evenOdd=False*)
Returns a boolean indicating if **(x, y)** is in the "black" area of the glyph.

postNotification(*notification*, *data=None*)
Post a **notification** through this object's notification dispatcher.

- **notification** The name of the notification.
- **data** Arbitrary data that will be stored in the `Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.postNotification(
    notification=notification, observable=anObject, data=data)
```

redo()
Perform a redo if possible, or return. If redo is performed, this will post *BaseObject.BeginRedo* and *BaseObject.EndRedo* notifications.

releaseHeldNotifications(*notification=None*)
Release this object's held notifications.

- **notification** The specific notification to hold. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.releaseHeldNotifications(
    observable=anObject, notification=notification)
```

removeAnchor(*anchor*)
Remove **anchor** from the glyph.

This will post a *Glyph.Changed* notification.

removeComponent(*component*)
Remove **component** from the glyph.

This will post a *Glyph.Changed* notification.

removeContour(*contour*)
Remove **contour** from the glyph.

This will post a *Glyph.Changed* notification.

removeGuideline(*guideline*)
Remove **guideline** from the glyph.

This will post a *Glyph.Changed* notification.

removeObserver(*observer*, *notification*)
Remove an observer from this object's notification dispatcher.

- **observer** A registered object.

- **notification** The notification that the observer was registered to be notified of.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.removeObserver(observer=observer,
    notification=notification, observable=anObject)
```

representationKeys()
Get a list of all representation keys that are currently cached.

rightMargin
The right margin of the glyph. Setting this posts *Glyph.WidthChanged*, *Glyph.RightMarginWillChange*, *Glyph.RightMarginDidChange* and *Glyph.Changed* notifications among others.

setDataFromSerialization(*data*)
Restore state from the provided data-dict.

tempLib
The glyph's `tempLib` object.

topMargin
The top margin of the glyph. Setting this posts *Glyph.HeightChanged*, *Glyph.VerticalOriginChanged*, *Glyph.TopMarginWillChange*, *Glyph.TopMarginDidChange* and *Glyph.Changed* notifications among others.

undo()
Perform an undo if possible, or return. If undo is performed, this will post *BaseObject.BeginUndo* and *BaseObject.EndUndo* notifications.

undoManager
The undo manager assigned to this object.

unicode
The primary unicode value for the glyph. This is the equivalent of `glyph.unicodes[0]`. This is a convenience attribute that works with the `unicodes` attribute.

unicodes
The list of unicode values assigned to the glyph. Setting this posts *Glyph.UnicodesChanged* and *Glyph.Changed* notifications.

verticalOrigin
The glyph's vertical origin. Setting this posts *Glyph.VerticalOriginChanged* and *Glyph.Changed* notifications.

width
The width of the glyph. Setting this posts *Glyph.WidthChanged* and *Glyph.Changed* notifications.

3.4 Contour

See also:

Notifications: The Contour object uses notifications to notify observers of changes.

3.4.1 Tasks

Reference Data

- *area*
- *bounds*
- *controlPointBounds*
- *open*

Direction

- *clockwise*
- *reverse()*

Points

- *Contour*
- *index()*
- *onCurvePoints*
- *setStartPoint()*

Segments

- *segments*
- *removeSegment()*
- *positionForProspectivePointInsertionAtSegmentAndT()*
- *splitAndInsertPointAtSegmentAndT()*

Hit Testing

- *contourInside()*
- *pointInside()*

Drawing

- *draw()*
- *drawPoints()*

Changed State

- *dirty*

Notifications

- *dispatcher*
- *addObserver()*
- *removeObserver()*
- *hasObserver()*

Parent

- `getParent()`
- `setParent()`

Contour

class defcon.**Contour**(*glyph=None*, *pointClass=None*)

This object represents a contour and it contains a list of points.

This object posts the following notifications:

- Contour.Changed
- Contour.WindingDirectionChanged
- Contour.PointsChanged
- Contour.IdentifierChanged

The Contour object has list like behavior. This behavior allows you to interact with point data directly. For example, to get a particular point:

```
point = contour[0]
```

To iterate over all points:

```
for point in contour:
```

To get the number of points:

```
pointCount = len(contour)
```

To interact with components or anchors in a similar way, use the `components` and `anchors` attributes.

addObserver(*observer*, *methodName*, *notification*, *identifier=None*)

Add an observer to this object's notification dispatcher.

- **observer** An object that can be referenced with weakref.
- **methodName** A string representing the method to be called when the notification is posted.
- **notification** The notification that the observer should be notified of.
- **identifier** None or a string identifying the observation. There is no requirement that the string be unique. A reverse domain naming scheme is recommended, but there are no requirements for the structure of the string.

The method that will be called as a result of the action must accept a single *notification* argument. This will be a `defcon.tools.notifications.Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.addObserver(observer=observer, methodName=methodName,
    notification=notification, observable=anObject, identifier=identifier)
```

addPoint(*values*, *segmentType=None*, *smooth=False*, *name=None*, *identifier=None*, _**kwargs_)
Standard point pen *addPoint* method. This should not be used externally.

appendPoint(*point*)
Append **point** to the glyph. The point must be a defcon `Point` object or a subclass of that object. An error will be raised if the point's identifier conflicts with any of the identifiers within the glyph.

This will post *Contour.PointsChanged* and *Contour.Changed* notifications.

area
The area of the contour's outline.

beginPath(*identifier=None*)
Standard point pen *beginPath* method. This should not be used externally.

bounds
The bounds of the contour's outline expressed as a tuple of form (xMin, yMin, xMax, yMax).

canRedo()
Returns a boolean indicating whether the undo manager is able to perform a redo.

canUndo()
Returns a boolean indicating whether the undo manager is able to perform an undo.

clear()
Clear the contents of the contour.

This posts *Contour.PointsChanged* and *Contour.Changed* notifications.

clockwise
A boolean representing if the contour has a clockwise direction. Setting this posts *Contour.WindingDirectionChanged* and *Contour.Changed* notifications.

contourInside(*other*, *segmentLength=10*)
Returns a boolean indicating if **other** is in the "black" area of the contour. This uses a flattened version of other's curves to calculate the location of the curves within this contour. **segmentLength** defines the desired length for the flattening process. A lower value will yeild higher accuracy but will require more computation time.

controlPointBounds
The control bounds of all points in the contour. This only measures the point positions, it does not measure curves. So, curves without points at the extrema will not be properly measured.

destroyAllRepresentations(*notification=None*)
Destroy all representations.

destroyRepresentation(*name*, _**kwargs_)
Destroy the stored representation for **name** and __**kwargs__. If no **kwargs** are given, any representation with **name** will be destroyed regardless of the **kwargs** passed when the representation was created.

dirty
The dirty state of the object. True if the object has been changed. False if not. Setting this to True will

cause the base changed notification to be posted. The object will automatically maintain this attribute and update it as you change the object.

disableNotifications(*notification=None*, *observer=None*)
Disable this object's notifications until told to resume them.

- **notification** The specific notification to disable. This is optional. If no *notification* is given, all notifications will be disabled.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.disableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

dispatcher
The `defcon.tools.notifications.NotificationCenter` assigned to the parent of this object.

draw(*pen*)
Draw the contour with **pen**.

drawPoints(*pointPen*)
Draw the contour with **pointPen**.

enableNotifications(*notification=None*, *observer=None*)
Enable this object's notifications.

- **notification** The specific notification to enable. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.enableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

endPath()
Standard point pen *endPath* method. This should not be used externally.

findObservations(*observer=None*, *notification=None*, *observable=None*, *identifier=None*)
Find observations of this object matching the given arguments based on the values that were passed during addObserver. A value of None for any of these indicates that all should be considered to match the value. In the case of identifier, strings will be matched using fnmatch.fnmatchcase. The returned value will be a list of dictionaries with this format:

[

{ observer=<. . . > observable=<. . . > methodName=". . . " notification=". . . " identifier=". . . "

}

]

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.findObservations(
    observer=observer, observable=anObject,
    notification=notification, identifier=identifier
)
```

font
The `Font` that this contour belongs to.

generateIdentifier()
Create a new, unique identifier for and assign it to the contour. This will post *Contour.IdentifierChanged* and *Contour.Changed* notifications.

generateIdentifierForPoint(*point*)
Create a new, unique identifier for and assign it to the point. This will post *Contour.Changed* notification.

getDataForSerialization(***kwargs*)
Return a dict of data that can be pickled.

getRepresentation(*name*, ***kwargs*)
Get a representation. **name** must be a registered representation name. ****kwargs** will be passed to the appropriate representation factory.

glyph
The `Glyph` that this contour belongs to. This should not be set externally.

hasCachedRepresentation(*name*, ***kwargs*)
Returns a boolean indicating if a representation for **name** and ****kwargs** is cached in the object.

hasObserver(*observer*, *notification*)
Returns a boolean indicating is the **observer** is registered for **notification**.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.hasObserver(observer=observer,
    notification=notification, observable=anObject)
```

holdNotifications(*notification=None*, *note=None*)
Hold this object's notifications until told to release them.

- **notification** The specific notification to hold. This is optional. If no *notification* is given, all notifications will be held.
- **note** An arbitrary string containing information about why the hold has been requested, the requester, etc. This is used for reference only.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.holdNotifications(
    observable=anObject, notification=notification, note=note)
```

identifier
The identifier. Setting this will post *Contour.IdentifierChanged* and *Contour.Changed* notifications.

identifiers
Set of identifiers for the glyph that this contour belongs to. This is primarily for internal use.

index(*point*)
Get the index for **point**.

insertPoint(*index*, *point*)
Insert **point** into the contour at index. The point must be a defcon `Point` object or a subclass of that object. An error will be raised if the points's identifier conflicts with any of the identifiers within the glyph.

This will post *Contour.PointsChanged* and *Contour.Changed* notifications.

layer
The `Layer` that this contour belongs to.

layerSet
The `LayerSet` that this contour belongs to.

move(*values*)
Move all points in the contour by **(x, y)**.

This will post *Contour.PointsChanged* and *Contour.Changed* notifications.

onCurvePoints
A list of all on curve points in the contour.

open
A boolean indicating if the contour is open or not.

pointClass
The class used for point.

pointInside(*coordinates*, *evenOdd=False*)
Returns a boolean indicating if **(x, y)** is in the "black" area of the contour.

positionForProspectivePointInsertionAtSegmentAndT(*segmentIndex*, *t*)
Get the precise coordinates and a boolean indicating if the point will be smooth for the given **segmentIndex** and **t**.

postNotification(*notification*, *data=None*)
Post a **notification** through this object's notification dispatcher.

- **notification** The name of the notification.
- **data** Arbitrary data that will be stored in the `Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.postNotification(
    notification=notification, observable=anObject, data=data)
```

redo()
Perform a redo if possible, or return. If redo is performed, this will post *BaseObject.BeginRedo* and *BaseObject.EndRedo* notifications.

releaseHeldNotifications(*notification=None*)
Release this object's held notifications.

- **notification** The specific notification to hold. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.releaseHeldNotifications(
    observable=anObject, notification=notification)
```

removeObserver(*observer*, *notification*)
Remove an observer from this object's notification dispatcher.

- **observer** A registered object.
- **notification** The notification that the observer was registered to be notified of.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.removeObserver(observer=observer,
    notification=notification, observable=anObject)
```

removePoint (*point*)
Remove **point** from the contour.

This will post *Contour.PointsChanged* and *Contour.Changed* notifications.

removeSegment (*segmentIndex*, *preserveCurve=False*)
Remove the segment at **segmentIndex**. If **preserveCurve** is True, the contour will try to preserve the overall curve shape.

representationKeys ()
Get a list of all representation keys that are currently cached.

reverse ()
Reverse the direction of the contour. It's important to note that the actual points stored in this object will be completely replaced by new points.

This will post *Contour.WindingDirectionChanged*, *Contour.PointsChanged* and *Contour.Changed* notifications.

segments
A list of all points in the contour organized into segments.

setDataFromSerialization (*data*)
Restore state from the provided data-dict.

setStartPoint (*index*)
Set the point at **index** as the first point in the contour. This point must be an on-curve point.

This will post *Contour.PointsChanged* and *Contour.Changed* notifications.

splitAndInsertPointAtSegmentAndT (*segmentIndex*, *t*)
Insert a point into the contour for the given **segmentIndex** and **t**.

This posts a *Contour.Changed* notification.

undo ()
Perform an undo if possible, or return. If undo is performed, this will post *BaseObject.BeginUndo* and *BaseObject.EndUndo* notifications.

undoManager
The undo manager assigned to this object.

3.5 Component

See also:

Notifications: The Component object uses notifications to notify observers of changes.

3.5.1 Tasks

Reference Data

- *bounds*

- *bounds*

Properties

- *baseGlyph*
- *transformation*

Hit Testing

- *pointInside()*

Drawing

- *draw()*
- *drawPoints()*

Changed State

- *dirty*

Notifications

- *dispatcher*
- *addObserver()*
- *removeObserver()*
- *hasObserver()*

Parent

- getParent()
- setParent()

Component

class defcon.**Component**(*glyph=None*)

This object represents a reference to another glyph.

This object posts the following notifications:

- Component.Changed
- Component.BaseGlyphChanged
- Component.BaseGlyphDataChanged
- Component.TransformationChanged
- Component.IdentifierChanged

addObserver(*observer*, *methodName*, *notification*, *identifier=None*)

Add an observer to this object's notification dispatcher.

- **observer** An object that can be referenced with weakref.
- **methodName** A string representing the method to be called when the notification is posted.
- **notification** The notification that the observer should be notified of.
- **identifier** None or a string identifying the observation. There is no requirement that the string be unique. A reverse domain naming scheme is recommended, but there are no requirements for the structure of the string.

The method that will be called as a result of the action must accept a single *notification* argument. This will be a `defcon.tools.notifications.Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.addObserver(observer=observer, methodName=methodName,
    notification=notification, observable=anObject, identifier=identifier)
```

baseGlyph

The glyph that the components references. Setting this will post *Component.BaseGlyphChanged* and *Component.Changed* notifications.

bounds

The bounds of the components's outline expressed as a tuple of form (xMin, yMin, xMax, yMax).

canRedo()

Returns a boolean indicating whether the undo manager is able to perform a redo.

canUndo()

Returns a boolean indicating whether the undo manager is able to perform an undo.

controlPointBounds

The control bounds of all points in the components. This only measures the point positions, it does not measure curves. So, curves without points at the extrema will not be properly measured.

destroyAllRepresentations(*notification=None*)

Destroy all representations.

destroyRepresentation(*name*, ***kwargs*)

Destroy the stored representation for **name** and ****kwargs**. If no **kwargs** are given, any representation with **name** will be destroyed regardless of the **kwargs** passed when the representation was created.

dirty

The dirty state of the object. True if the object has been changed. False if not. Setting this to True will cause the base changed notification to be posted. The object will automatically maintain this attribute and update it as you change the object.

disableNotifications(*notification=None*, *observer=None*)

Disable this object's notifications until told to resume them.

- **notification** The specific notification to disable. This is optional. If no *notification* is given, all notifications will be disabled.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.disableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

dispatcher
The `defcon.tools.notifications.NotificationCenter` assigned to the parent of this object.

draw(*pen*)
Draw the component with **pen**.

drawPoints(*pointPen*)
Draw the component with **pointPen**.

enableNotifications(*notification=None*, *observer=None*)
Enable this object's notifications.

- **notification** The specific notification to enable. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.enableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

findObservations(*observer=None*, *notification=None*, *observable=None*, *identifier=None*)
Find observations of this object matching the given arguments based on the values that were passed during addObserver. A value of None for any of these indicates that all should be considered to match the value. In the case of identifier, strings will be matched using fnmatch.fnmatchcase. The returned value will be a list of dictionaries with this format:

[

{ observer=<...> observable=<...> methodName="..." notification="..." identifier="..."

}

]

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.findObservations(
    observer=observer, observable=anObject,
    notification=notification, identifier=identifier
)
```

font
The `Font` that this component belongs to.

generateIdentifier()
Create a new, unique identifier for and assign it to the contour. This will post *Component.IdentifierChanged* and *Component.Changed* notifications.

getDataForSerialization(***kwargs*)
Return a dict of data that can be pickled.

getRepresentation(*name*, ***kwargs*)
Get a representation. **name** must be a registered representation name. ****kwargs** will be passed to the appropriate representation factory.

glyph
The `Glyph` that this component belongs to. This should not be set externally.

hasCachedRepresentation(*name*, ***kwargs*)
Returns a boolean indicating if a representation for **name** and ****kwargs** is cached in the object.

hasObserver(*observer*, *notification*)

Returns a boolean indicating is the **observer** is registered for **notification**.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.hasObserver(observer=observer,
    notification=notification, observable=anObject)
```

holdNotifications(*notification=None*, *note=None*)

Hold this object's notifications until told to release them.

- **notification** The specific notification to hold. This is optional. If no *notification* is given, all notifications will be held.
- **note** An arbitrary string containing information about why the hold has been requested, the requester, etc. This is used for reference only.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.holdNotifications(
    observable=anObject, notification=notification, note=note)
```

identifier

The identifier. Setting this will post *Component.IdentifierChanged* and *Component.Changed* notifications.

identifiers

Set of identifiers for the glyph that this component belongs to. This is primarily for internal use.

layer

The `Layer` that this component belongs to.

layerSet

The `LayerSet` that this component belongs to.

move(*values*)

Move the component by **(x, y)**.

This posts *Component.TransformationChanged* and *Component.Changed* notifications.

pointInside(*coordinates*, *evenOdd=False*)

Returns a boolean indicating if **(x, y)** is in the "black" area of the component.

postNotification(*notification*, *data=None*)

Post a **notification** through this object's notification dispatcher.

- **notification** The name of the notification.
- **data** Arbitrary data that will be stored in the `Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.postNotification(
    notification=notification, observable=anObject, data=data)
```

redo()

Perform a redo if possible, or return. If redo is performed, this will post *BaseObject.BeginRedo* and *BaseObject.EndRedo* notifications.

releaseHeldNotifications(*notification=None*)

Release this object's held notifications.

- **notification** The specific notification to hold. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.releaseHeldNotifications(
    observable=anObject, notification=notification)
```

removeObserver(*observer*, *notification*)
Remove an observer from this object's notification dispatcher.

- **observer** A registered object.
- **notification** The notification that the observer was registered to be notified of.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.removeObserver(observer=observer,
    notification=notification, observable=anObject)
```

representationKeys()
Get a list of all representation keys that are currently cached.

setDataFromSerialization(*data*)
Restore state from the provided data-dict.

transformation
The transformation matrix for the component. Setting this will post *Component.TransformationChanged* and *Component.Changed* notifications.

undo()
Perform an undo if possible, or return. If undo is performed, this will post *BaseObject.BeginUndo* and *BaseObject.EndUndo* notifications.

undoManager
The undo manager assigned to this object.

3.6 Point

Note: This object is not a subclass of `BaseObject` and therefore it does not produce notifications or have any parent attributes. This may change in the future.

3.6.1 Tasks

Position

- *x*
- *y*

Type

- *segmentType*
- *smooth*

Move

- *move*

Point

class defcon.**Point**(*coordinates*, *segmentType=None*, *smooth=False*, *name=None*, *identifier=None*)
This object represents a single point.

identifier
The identifier.

move(*values*)
Move the component by **(x, y)**.

name
An arbitrary name for the point.

segmentType
The segment type. The positibilies are *move*, *line*, *curve*, *qcurve* and *None* (indicating that this is an off-curve point).

smooth
A boolean indicating the smooth state of the point.

x
The x coordinate.

y
The y coordinate.

3.7 Anchor

See also:

Notifications: The Anchor object uses notifications to notify observers of changes.

3.7.1 Tasks

Position

- *x*
- *y*

Name

- *name*

Move

- `move`

Notifications

- `dispatcher`
- `addObserver()`
- `removeObserver()`
- `hasObserver()`

Parent

- `getParent()`
- `setParent()`

Anchor

class `defcon.`**`Anchor`**(*glyph=None*, *anchorDict=None*)

This object represents an anchor point.

This object posts the following notifications:

- Anchor.Changed
- Anchor.XChanged
- Anchor.YChanged
- Anchor.NameChanged
- Anchor.ColorChanged
- Anchor.IdentifierChanged

During initialization an anchor dictionary can be passed. If so, the new object will be populated with the data from the dictionary.

`addObserver`(*observer*, *methodName*, *notification*, *identifier=None*)

Add an observer to this object's notification dispatcher.

- **observer** An object that can be referenced with weakref.
- **methodName** A string representing the method to be called when the notification is posted.
- **notification** The notification that the observer should be notified of.
- **identifier** None or a string identifying the observation. There is no requirement that the string be unique. A reverse domain naming scheme is recommended, but there are no requirements for the structure of the string.

The method that will be called as a result of the action must accept a single *notification* argument. This will be a `defcon.tools.notifications.Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.addObserver(observer=observer, methodName=methodName,
    notification=notification, observable=anObject, identifier=identifier)
```

canRedo()
Returns a boolean indicating whether the undo manager is able to perform a redo.

canUndo()
Returns a boolean indicating whether the undo manager is able to perform an undo.

clear() → None. Remove all items from D.

color
The anchors's `Color` object. When setting, the value can be a UFO color string, a sequence of (r, g, b, a) or a `Color` object. Setting this posts *Anchor.ColorChanged* and *Anchor.Changed* notifications.

copy() → a shallow copy of D

destroyAllRepresentations(*notification=None*)
Destroy all representations.

destroyRepresentation(*name*, ***kwargs*)
Destroy the stored representation for **name** and ****kwargs**. If no **kwargs** are given, any representation with **name** will be destroyed regardless of the **kwargs** passed when the representation was created.

dirty
The dirty state of the object. True if the object has been changed. False if not. Setting this to True will cause the base changed notification to be posted. The object will automatically maintain this attribute and update it as you change the object.

disableNotifications(*notification=None*, *observer=None*)
Disable this object's notifications until told to resume them.

- **notification** The specific notification to disable. This is optional. If no *notification* is given, all notifications will be disabled.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.disableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

dispatcher
The *defcon.tools.notifications.NotificationCenter* assigned to the parent of this object.

enableNotifications(*notification=None*, *observer=None*)
Enable this object's notifications.

- **notification** The specific notification to enable. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.enableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

findObservations(*observer=None*, *notification=None*, *observable=None*, *identifier=None*)
Find observations of this object matching the given arguments based on the values that were passed during addObserver. A value of None for any of these indicates that all should be considered to match the value.

In the case of identifier, strings will be matched using fnmatch.fnmatchcase. The returned value will be a list of dictionaries with this format:

[

{ observer=<…> observable=<…> methodName="…" notification="…" identifier="…"

}

]

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.findObservations(
    observer=observer, observable=anObject,
    notification=notification, identifier=identifier
)
```

font
The `Font` that this anchor belongs to.

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

generateIdentifier()
Create a new, unique identifier for and assign it to the guideline. This will post *Anchor.IdentifierChanged* and *Anchor.Changed* notifications.

get()
Return the value for key if key is in the dictionary, else default.

getDataForSerialization(***kwargs*)
Return a dict of data that can be pickled.

getRepresentation(*name*, ***kwargs*)
Get a representation. **name** must be a registered representation name. ****kwargs** will be passed to the appropriate representation factory.

glyph
The `Glyph` that this anchor belongs to. This should not be set externally.

hasCachedRepresentation(*name*, ***kwargs*)
Returns a boolean indicating if a representation for **name** and ****kwargs** is cached in the object.

hasObserver(*observer*, *notification*)
Returns a boolean indicating is the **observer** is registered for **notification**.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.hasObserver(observer=observer,
    notification=notification, observable=anObject)
```

holdNotifications(*notification=None*, *note=None*)
Hold this object's notifications until told to release them.

- **notification** The specific notification to hold. This is optional. If no *notification* is given, all notifications will be held.
- **note** An arbitrary string containing information about why the hold has been requested, the requester, etc. This is used for reference only.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.holdNotifications(
    observable=anObject, notification=notification, note=note)
```

identifier
The identifier. Setting this will post *Anchor.IdentifierChanged* and *Anchor.Changed* notifications.

identifiers
Set of identifiers for the glyph that this anchor belongs to. This is primarily for internal use.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

layer
The `Layer` that this anchor belongs to.

layerSet
The `LayerSet` that this anchor belongs to.

move(*values*)
Move the anchor by **(x, y)**.

This will post *Anchor.XChange*, *Anchor.YChanged* and *Anchor.Changed* notifications if anything changed.

name
The name. Setting this will post *Anchor.NameChanged* and *Anchor.Changed* notifications.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

postNotification(*notification*, *data=None*)
Post a **notification** through this object's notification dispatcher.

- **notification** The name of the notification.
- **data** Arbitrary data that will be stored in the `Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.postNotification(
    notification=notification, observable=anObject, data=data)
```

redo()
Perform a redo if possible, or return. If redo is performed, this will post *BaseObject.BeginRedo* and *BaseObject.EndRedo* notifications.

releaseHeldNotifications(*notification=None*)
Release this object's held notifications.

- **notification** The specific notification to hold. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.releaseHeldNotifications(
    observable=anObject, notification=notification)
```

removeObserver(*observer*, *notification*)
Remove an observer from this object's notification dispatcher.

- **observer** A registered object.
- **notification** The notification that the observer was registered to be notified of.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.removeObserver(observer=observer,
    notification=notification, observable=anObject)
```

representationKeys()
Get a list of all representation keys that are currently cached.

setDataFromSerialization(*data*)
Restore state from the provided data-dict.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

undo()
Perform an undo if possible, or return. If undo is performed, this will post *BaseObject.BeginUndo* and *BaseObject.EndUndo* notifications.

undoManager
The undo manager assigned to this object.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

x
The x coordinate. Setting this will post *Anchor.XChanged* and *Anchor.Changed* notifications.

y
The y coordinate. Setting this will post *Anchor.YChanged* and *Anchor.Changed* notifications.

3.8 Info

See also:

Notifications: The Info object uses notifications to notify observers of changes.

3.8.1 Tasks

Generic Identification

- `familyName`
- `styleName`
- `styleMapFamilyName`
- `styleMapStyleName`

- `versionMajor`
- `versionMinor`
- `year`

Generic Legal

- `copyright`
- `trademark`

Generic Dimensions

- `unitsPerEm`
- `descender`
- `xHeight`
- `capHeight`
- `ascender`
- `italicAngle`

Generic Miscellaneous

- `note`

OpenType head Table

- `openTypeHeadCreated`
- `openTypeHeadLowestRecPPEM`
- `openTypeHeadFlags`

OpenType hhea Table

- `openTypeHheaAscender`
- `openTypeHheaDescender`
- `openTypeHheaLineGap`
- `openTypeHheaCaretSlopeRise`
- `openTypeHheaCaretSlopeRun`
- `openTypeHheaCaretOffset`

OpenType name Table

- `openTypeNameDesigner`
- `openTypeNameDesignerURL`
- `openTypeNameManufacturer`
- `openTypeNameManufacturerURL`
- `openTypeNameLicense`
- `openTypeNameLicenseURL`
- `openTypeNameVersion`
- `openTypeNameUniqueID`
- `openTypeNameDescription`
- `openTypeNamePreferredFamilyName`
- `openTypeNamePreferredSubfamilyName`
- `openTypeNameCompatibleFullName`
- `openTypeNameSampleText`
- `openTypeNameWWSFamilyName`
- `openTypeNameWWSSubfamilyName`

OpenType OS/2 Table

- `openTypeOS2WidthClass`
- `openTypeOS2WeightClass`
- `openTypeOS2Selection`
- `openTypeOS2VendorID`
- `openTypeOS2Panose`
- `openTypeOS2FamilyClass`
- `openTypeOS2UnicodeRanges`
- `openTypeOS2CodePageRanges`
- `openTypeOS2TypoAscender`
- `openTypeOS2TypoDescender`
- `openTypeOS2TypoLineGap`
- `openTypeOS2WinAscent`
- `openTypeOS2WinDescent`
- `openTypeOS2Type`
- `openTypeOS2SubscriptXSize`
- `openTypeOS2SubscriptYSize`
- `openTypeOS2SubscriptXOffset`
- `openTypeOS2SubscriptYOffset`

- `openTypeOS2SuperscriptXSize`
- `openTypeOS2SuperscriptYSize`
- `openTypeOS2SuperscriptXOffset`
- `openTypeOS2SuperscriptYOffset`
- `openTypeOS2StrikeoutSize`
- `openTypeOS2StrikeoutPosition`
- `openTypeVheaVertTypoAscender`
- `openTypeVheaVertTypoDescender`
- `openTypeVheaVertTypoLineGap`
- `openTypeVheaCaretSlopeRise`
- `openTypeVheaCaretSlopeRun`
- `openTypeVheaCaretOffset`

Postscript

- `postscriptFontName`
- `postscriptFullName`
- `postscriptSlantAngle`
- `postscriptUniqueID`
- `postscriptUnderlineThickness`
- `postscriptUnderlinePosition`
- `postscriptIsFixedPitch`
- `postscriptBlueValues`
- `postscriptOtherBlues`
- `postscriptFamilyBlues`
- `postscriptFamilyOtherBlues`
- `postscriptStemSnapH`
- `postscriptStemSnapV`
- `postscriptBlueFuzz`
- `postscriptBlueShift`
- `postscriptBlueScale`
- `postscriptForceBold`
- `postscriptDefaultWidthX`
- `postscriptNominalWidthX`
- `postscriptWeightName`
- `postscriptDefaultCharacter`
- `postscriptWindowsCharacterSet`

Macintosh FOND Resource

- `macintoshFONDFamilyID`
- `macintoshFONDName`

Info

class `defcon.`**`Info`**(*font=None*, *guidelineClass=None*)

This object represents info values.

This object posts the following notifications:

- Info.Changed
- Info.BeginUndo
- Info.EndUndo
- Info.BeginRedo
- Info.EndRedo
- Info.ValueChanged

Note: The documentation strings here were automatically generated from the UFO specification.

`addObserver`(*observer*, *methodName*, *notification*, *identifier=None*)

Add an observer to this object's notification dispatcher.

- **observer** An object that can be referenced with weakref.
- **methodName** A string representing the method to be called when the notification is posted.
- **notification** The notification that the observer should be notified of.
- **identifier** None or a string identifying the observation. There is no requirement that the string be unique. A reverse domain naming scheme is recommended, but there are no requirements for the structure of the string.

The method that will be called as a result of the action must accept a single *notification* argument. This will be a *`defcon.tools.notifications.Notification`* object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.addObserver(observer=observer, methodName=methodName,
    notification=notification, observable=anObject, identifier=identifier)
```

`canRedo`()

Returns a boolean indicating whether the undo manager is able to perform a redo.

`canUndo`()

Returns a boolean indicating whether the undo manager is able to perform an undo.

`destroyAllRepresentations`(*notification=None*)

Destroy all representations.

`destroyRepresentation`(*name*, ***kwargs*)

Destroy the stored representation for **name** and ****kwargs**. If no **kwargs** are given, any representation with **name** will be destroyed regardless of the **kwargs** passed when the representation was created.

dirty
The dirty state of the object. True if the object has been changed. False if not. Setting this to True will cause the base changed notification to be posted. The object will automatically maintain this attribute and update it as you change the object.

disableNotifications(*notification=None*, *observer=None*)
Disable this object's notifications until told to resume them.

- **notification** The specific notification to disable. This is optional. If no *notification* is given, all notifications will be disabled.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.disableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

dispatcher
The `defcon.tools.notifications.NotificationCenter` assigned to the parent of this object.

enableNotifications(*notification=None*, *observer=None*)
Enable this object's notifications.

- **notification** The specific notification to enable. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.enableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

findObservations(*observer=None*, *notification=None*, *observable=None*, *identifier=None*)
Find observations of this object matching the given arguments based on the values that were passed during addObserver. A value of None for any of these indicates that all should be considered to match the value. In the case of identifier, strings will be matched using fnmatch.fnmatchcase. The returned value will be a list of dictionaries with this format:

[

{ observer=<…> observable=<…> methodName="…" notification="…" identifier="…"

}

]

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.findObservations(
    observer=observer, observable=anObject,
    notification=notification, identifier=identifier
)
```

font
The `Font` that this object belongs to.

getDataForSerialization(***kwargs*)
Return a dict of data that can be pickled.

getRepresentation(*name*, ***kwargs*)
Get a representation. **name** must be a registered representation name. ****kwargs** will be passed to the appropriate representation factory.

guidelines
This is a compatibility attribute for ufoLib. It maps to `Font.guidelines`.

hasCachedRepresentation(*name*, ***kwargs*)
Returns a boolean indicating if a representation for **name** and ****kwargs** is cached in the object.

hasObserver(*observer*, *notification*)
Returns a boolean indicating is the **observer** is registered for **notification**.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.hasObserver(observer=observer,
    notification=notification, observable=anObject)
```

holdNotifications(*notification=None*, *note=None*)
Hold this object's notifications until told to release them.

- **notification** The specific notification to hold. This is optional. If no *notification* is given, all notifications will be held.
- **note** An arbitrary string containing information about why the hold has been requested, the requester, etc. This is used for reference only.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.holdNotifications(
    observable=anObject, notification=notification, note=note)
```

postNotification(*notification*, *data=None*)
Post a **notification** through this object's notification dispatcher.

- **notification** The name of the notification.
- **data** Arbitrary data that will be stored in the `Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.postNotification(
    notification=notification, observable=anObject, data=data)
```

redo()
Perform a redo if possible, or return. If redo is performed, this will post *BaseObject.BeginRedo* and *BaseObject.EndRedo* notifications.

releaseHeldNotifications(*notification=None*)
Release this object's held notifications.

- **notification** The specific notification to hold. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.releaseHeldNotifications(
    observable=anObject, notification=notification)
```

removeObserver(*observer*, *notification*)
Remove an observer from this object's notification dispatcher.

- **observer** A registered object.
- **notification** The notification that the observer was registered to be notified of.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.removeObserver(observer=observer,
    notification=notification, observable=anObject)
```

representationKeys()
Get a list of all representation keys that are currently cached.

setDataFromSerialization(*data*)
Restore state from the provided data-dict.

undo()
Perform an undo if possible, or return. If undo is performed, this will post *BaseObject.BeginUndo* and *BaseObject.EndUndo* notifications.

undoManager
The undo manager assigned to this object.

3.9 Kerning

See also:

Notifications: The Kerning object uses notifications to notify observers of changes.

3.9.1 Tasks

Notifications

- `dispatcher`
- `addObserver()`
- `removeObserver()`
- `hasObserver()`

Parent

- `getParent()`
- `setParent()`

Kerning

class defcon.**Kerning**(*font=None*)
This object contains all of the kerning pairs in a font.

This object posts the following notifications:

- Kerning.Changed
- Kerning.BeginUndo
- Kerning.EndUndo
- Kerning.BeginRedo
- Kerning.EndRedo
- Kerning.PairSet
- Kerning.PairDeleted
- Kerning.Cleared
- Kerning.Updated

This object behaves like a dict. For example, to get a list of all kerning pairs:

```
pairs = kerning.keys()
```

To get all pairs including the values:

```
for (left, right), value in kerning.items():
```

To get the value for a particular pair:

```
value = kerning["a", "b"]
```

To set the value for a particular pair:

```
kerning["a", "b"] = 100
```

And so on.

Note: This object is not very smart in the way it handles zero values, exceptions, etc. This may change in the future.

addObserver(*observer*, *methodName*, *notification*, *identifier=None*)
Add an observer to this object's notification dispatcher.

- **observer** An object that can be referenced with weakref.
- **methodName** A string representing the method to be called when the notification is posted.
- **notification** The notification that the observer should be notified of.
- **identifier** None or a string identifying the observation. There is no requirement that the string be unique. A reverse domain naming scheme is recommended, but there are no requirements for the structure of the string.

The method that will be called as a result of the action must accept a single *notification* argument. This will be a *defcon.tools.notifications.Notification* object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.addObserver(observer=observer, methodName=methodName,
    notification=notification, observable=anObject, identifier=identifier)
```

canRedo()
Returns a boolean indicating whether the undo manager is able to perform a redo.

canUndo()
Returns a boolean indicating whether the undo manager is able to perform an undo.

clear() → None. Remove all items from D.

copy() → a shallow copy of D

destroyAllRepresentations(*notification=None*)
Destroy all representations.

destroyRepresentation(*name*, ***kwargs*)
Destroy the stored representation for **name** and ****kwargs**. If no **kwargs** are given, any representation with **name** will be destroyed regardless of the **kwargs** passed when the representation was created.

dirty
The dirty state of the object. True if the object has been changed. False if not. Setting this to True will cause the base changed notification to be posted. The object will automatically maintain this attribute and update it as you change the object.

disableNotifications(*notification=None*, *observer=None*)
Disable this object's notifications until told to resume them.

- **notification** The specific notification to disable. This is optional. If no *notification* is given, all notifications will be disabled.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.disableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

dispatcher
The `defcon.tools.notifications.NotificationCenter` assigned to the parent of this object.

enableNotifications(*notification=None*, *observer=None*)
Enable this object's notifications.

- **notification** The specific notification to enable. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.enableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

find(*pair*, *default=0*)
This will find the value for **pair** even if **pair** is not specifically defined. For example: You have a group named *public.kern1.A* with the contents *["A", "Aacute"]* and you have a group named *public.kern2.C* with the contents *["C", "Ccedilla"]*. The only defined kerning is *("public.kern1.A", public.kern2.C) = 100*. If you use this method to find the value for *("A", "Ccedilla")* you will get *100*.

findObservations(*observer=None*, *notification=None*, *observable=None*, *identifier=None*)
Find observations of this object matching the given arguments based on the values that were passed during addObserver. A value of None for any of these indicates that all should be considered to match the value. In the case of identifier, strings will be matched using fnmatch.fnmatchcase. The returned value will be a list of dictionaries with this format:

[

{ observer=<...> observable=<...> methodName="..." notification="..." identifier="..."

}

]

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.findObservations(
    observer=observer, observable=anObject,
    notification=notification, identifier=identifier
)
```

font
The *Font* that this object belongs to.

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

get(*pair*, *default=0*)
Return the value for key if key is in the dictionary, else default.

getDataForSerialization(***kwargs*)
Return a dict of data that can be pickled.

getRepresentation(*name*, ***kwargs*)
Get a representation. **name** must be a registered representation name. ****kwargs** will be passed to the appropriate representation factory.

hasCachedRepresentation(*name*, ***kwargs*)
Returns a boolean indicating if a representation for **name** and ****kwargs** is cached in the object.

hasObserver(*observer*, *notification*)
Returns a boolean indicating is the **observer** is registered for **notification**.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.hasObserver(observer=observer,
    notification=notification, observable=anObject)
```

holdNotifications(*notification=None*, *note=None*)
Hold this object's notifications until told to release them.

- **notification** The specific notification to hold. This is optional. If no *notification* is given, all notifications will be held.
- **note** An arbitrary string containing information about why the hold has been requested, the requester, etc. This is used for reference only.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.holdNotifications(
    observable=anObject, notification=notification, note=note)
```

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

postNotification(*notification*, *data=None*)
Post a **notification** through this object's notification dispatcher.

- **notification** The name of the notification.
- **data** Arbitrary data that will be stored in the `Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.postNotification(
    notification=notification, observable=anObject, data=data)
```

redo()
Perform a redo if possible, or return. If redo is performed, this will post *BaseObject.BeginRedo* and *BaseObject.EndRedo* notifications.

releaseHeldNotifications(*notification=None*)
Release this object's held notifications.

- **notification** The specific notification to hold. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.releaseHeldNotifications(
    observable=anObject, notification=notification)
```

removeObserver(*observer*, *notification*)
Remove an observer from this object's notification dispatcher.

- **observer** A registered object.
- **notification** The notification that the observer was registered to be notified of.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.removeObserver(observer=observer,
    notification=notification, observable=anObject)
```

representationKeys()
Get a list of all representation keys that are currently cached.

setDataFromSerialization(*data*)
Restore state from the provided data-dict.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

undo()
Perform an undo if possible, or return. If undo is performed, this will post *BaseObject.BeginUndo* and *BaseObject.EndUndo* notifications.

undoManager
The undo manager assigned to this object.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

3.10 Groups

See also:

Notifications: The Groups object uses notifications to notify observers of changes.

3.10.1 Tasks

Notifications

- *dispatcher*
- *addObserver()*
- *removeObserver()*
- *hasObserver()*

Parent

- getParent()
- setParent()

Groups

class defcon.**Groups**(*font=None*)
This object contains all of the groups in a font.

This object posts the following notifications:

- Groups.Changed
- Groups.BeginUndo
- Groups.EndUndo
- Groups.BeginRedo
- Groups.EndRedo
- Groups.GroupSet
- Groups.GroupDeleted
- Groups.Cleared
- Groups.Updated

This object behaves like a dict. The keys are group names and the values are lists of glyph names:

```
{
    "myGroup" : ["a", "b"],
    "myOtherGroup" : ["a.alt", "g.alt"],
}
```

The API for interacting with the data is the same as a standard dict. For example, to get a list of all group names:

```
groupNames = groups.keys()
```

To get all groups including the glyph lists:

```
for groupName, glyphList in groups.items():
```

To get the glyph list for a particular group name:

```
glyphList = groups["myGroup"]
```

To set the glyph list for a particular group name:

```
groups["myGroup"] = ["x", "y", "z"]
```

And so on.

Note: You should not modify the group list and expect the object to know about it. For example, this could cause your changes to be lost:

```
glyphList = groups["myGroups"]
glyphList.append("n")
```

To make sure the change is noticed, reset the list into the object:

```
glyphList = groups["myGroups"]
glyphList.append("n")
groups["myGroups"] = glyphList
```

This may change in the future.

addObserver(*observer*, *methodName*, *notification*, *identifier=None*)

Add an observer to this object's notification dispatcher.

- **observer** An object that can be referenced with weakref.
- **methodName** A string representing the method to be called when the notification is posted.
- **notification** The notification that the observer should be notified of.
- **identifier** None or a string identifying the observation. There is no requirement that the string be unique. A reverse domain naming scheme is recommended, but there are no requirements for the structure of the string.

The method that will be called as a result of the action must accept a single *notification* argument. This will be a *defcon.tools.notifications.Notification* object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.addObserver(observer=observer, methodName=methodName,
    notification=notification, observable=anObject, identifier=identifier)
```

canRedo()
Returns a boolean indicating whether the undo manager is able to perform a redo.

canUndo()
Returns a boolean indicating whether the undo manager is able to perform an undo.

clear() → None. Remove all items from D.

copy() → a shallow copy of D

destroyAllRepresentations(*notification=None*)
Destroy all representations.

destroyRepresentation(*name*, ***kwargs*)
Destroy the stored representation for **name** and ****kwargs**. If no **kwargs** are given, any representation with **name** will be destroyed regardless of the **kwargs** passed when the representation was created.

dirty
The dirty state of the object. True if the object has been changed. False if not. Setting this to True will cause the base changed notification to be posted. The object will automatically maintain this attribute and update it as you change the object.

disableNotifications(*notification=None*, *observer=None*)
Disable this object's notifications until told to resume them.

- **notification** The specific notification to disable. This is optional. If no *notification* is given, all notifications will be disabled.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.disableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

dispatcher
The *defcon.tools.notifications.NotificationCenter* assigned to the parent of this object.

enableNotifications(*notification=None*, *observer=None*)
Enable this object's notifications.

- **notification** The specific notification to enable. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.enableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

findObservations(*observer=None*, *notification=None*, *observable=None*, *identifier=None*)
Find observations of this object matching the given arguments based on the values that were passed during addObserver. A value of None for any of these indicates that all should be considered to match the value. In the case of identifier, strings will be matched using fnmatch.fnmatchcase. The returned value will be a list of dictionaries with this format:

[

{ observer=<...> observable=<...> methodName="..." notification="..." identifier="..."

}

]

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.findObservations(
    observer=observer, observable=anObject,
    notification=notification, identifier=identifier
)
```

font
The `Font` that this object belongs to.

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

get()
Return the value for key if key is in the dictionary, else default.

getDataForSerialization(***kwargs*)
Return a dict of data that can be pickled.

getRepresentation(*name*, ***kwargs*)
Get a representation. **name** must be a registered representation name. ****kwargs** will be passed to the appropriate representation factory.

hasCachedRepresentation(*name*, ***kwargs*)
Returns a boolean indicating if a representation for **name** and ****kwargs** is cached in the object.

hasObserver(*observer*, *notification*)
Returns a boolean indicating is the **observer** is registered for **notification**.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.hasObserver(observer=observer,
    notification=notification, observable=anObject)
```

holdNotifications(*notification=None*, *note=None*)
Hold this object's notifications until told to release them.

- **notification** The specific notification to hold. This is optional. If no *notification* is given, all notifications will be held.
- **note** An arbitrary string containing information about why the hold has been requested, the requester, etc. This is used for reference only.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.holdNotifications(
    observable=anObject, notification=notification, note=note)
```

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

postNotification(*notification*, *data=None*)
Post a **notification** through this object's notification dispatcher.

- **notification** The name of the notification.
- **data** Arbitrary data that will be stored in the `Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.postNotification(
    notification=notification, observable=anObject, data=data)
```

redo()
Perform a redo if possible, or return. If redo is performed, this will post *BaseObject.BeginRedo* and *BaseObject.EndRedo* notifications.

releaseHeldNotifications(*notification=None*)
Release this object's held notifications.

- **notification** The specific notification to hold. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.releaseHeldNotifications(
    observable=anObject, notification=notification)
```

removeObserver(*observer*, *notification*)
Remove an observer from this object's notification dispatcher.

- **observer** A registered object.
- **notification** The notification that the observer was registered to be notified of.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.removeObserver(observer=observer,
    notification=notification, observable=anObject)
```

representationKeys()
Get a list of all representation keys that are currently cached.

setDataFromSerialization(*data*)
Restore state from the provided data-dict.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

undo()
Perform an undo if possible, or return. If undo is performed, this will post *BaseObject.BeginUndo* and *BaseObject.EndUndo* notifications.

undoManager
The undo manager assigned to this object.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

3.11 Features

See also:

Notifications: The Features object uses notifications to notify observers of changes.

3.11.1 Tasks

Feature Text

- *text*

Notifications

- *dispatcher*
- *addObserver()*
- *removeObserver()*
- *hasObserver()*

Parent

- getParent()
- setParent()

Features

class defcon.**Features**(*font=None*)

This object contais the test represening features in the font.

This object posts the following notifications:

- Features.Changed
- Features.BeginUndo
- Features.EndUndo
- Features.BeginRedo
- Features.EndRedo
- Features.TextChanged

addObserver(*observer*, *methodName*, *notification*, *identifier=None*)

Add an observer to this object's notification dispatcher.

- **observer** An object that can be referenced with weakref.
- **methodName** A string representing the method to be called when the notification is posted.
- **notification** The notification that the observer should be notified of.

- **identifier** None or a string identifying the observation. There is no requirement that the string be unique. A reverse domain naming scheme is recommended, but there are no requirements for the structure of the string.

The method that will be called as a result of the action must accept a single *notification* argument. This will be a `defcon.tools.notifications.Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.addObserver(observer=observer, methodName=methodName,
    notification=notification, observable=anObject, identifier=identifier)
```

canRedo()
Returns a boolean indicating whether the undo manager is able to perform a redo.

canUndo()
Returns a boolean indicating whether the undo manager is able to perform an undo.

destroyAllRepresentations(*notification=None*)
Destroy all representations.

destroyRepresentation(*name*, ***kwargs*)
Destroy the stored representation for **name** and ****kwargs**. If no **kwargs** are given, any representation with **name** will be destroyed regardless of the **kwargs** passed when the representation was created.

dirty
The dirty state of the object. True if the object has been changed. False if not. Setting this to True will cause the base changed notification to be posted. The object will automatically maintain this attribute and update it as you change the object.

disableNotifications(*notification=None*, *observer=None*)
Disable this object's notifications until told to resume them.

- **notification** The specific notification to disable. This is optional. If no *notification* is given, all notifications will be disabled.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.disableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

dispatcher
The `defcon.tools.notifications.NotificationCenter` assigned to the parent of this object.

enableNotifications(*notification=None*, *observer=None*)
Enable this object's notifications.

- **notification** The specific notification to enable. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.enableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

findObservations(*observer=None*, *notification=None*, *observable=None*, *identifier=None*)
Find observations of this object matching the given arguments based on the values that were passed during addObserver. A value of None for any of these indicates that all should be considered to match the value.

In the case of identifier, strings will be matched using fnmatch.fnmatchcase. The returned value will be a list of dictionaries with this format:

[

{ observer=<…> observable=<…> methodName="…" notification="…" identifier="…"

}

]

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.findObservations(
    observer=observer, observable=anObject,
    notification=notification, identifier=identifier
)
```

`font`
The *`Font`* that this object belongs to.

`getDataForSerialization`(***kwargs*)
Return a dict of data that can be pickled.

`getRepresentation`(*name*, ***kwargs*)
Get a representation. **name** must be a registered representation name. ****kwargs** will be passed to the appropriate representation factory.

`hasCachedRepresentation`(*name*, ***kwargs*)
Returns a boolean indicating if a representation for **name** and ****kwargs** is cached in the object.

`hasObserver`(*observer*, *notification*)
Returns a boolean indicating is the **observer** is registered for **notification**.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.hasObserver(observer=observer,
    notification=notification, observable=anObject)
```

`holdNotifications`(*notification=None*, *note=None*)
Hold this object's notifications until told to release them.

- **notification** The specific notification to hold. This is optional. If no *notification* is given, all notifications will be held.
- **note** An arbitrary string containing information about why the hold has been requested, the requester, etc. This is used for reference only.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.holdNotifications(
    observable=anObject, notification=notification, note=note)
```

`postNotification`(*notification*, *data=None*)
Post a **notification** through this object's notification dispatcher.

- **notification** The name of the notification.
- **data** Arbitrary data that will be stored in the `Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.postNotification(
    notification=notification, observable=anObject, data=data)
```

redo()
Perform a redo if possible, or return. If redo is performed, this will post *BaseObject.BeginRedo* and *BaseObject.EndRedo* notifications.

releaseHeldNotifications(*notification=None*)
Release this object's held notifications.

- **notification** The specific notification to hold. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.releaseHeldNotifications(
    observable=anObject, notification=notification)
```

removeObserver(*observer*, *notification*)
Remove an observer from this object's notification dispatcher.

- **observer** A registered object.
- **notification** The notification that the observer was registered to be notified of.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.removeObserver(observer=observer,
    notification=notification, observable=anObject)
```

representationKeys()
Get a list of all representation keys that are currently cached.

setDataFromSerialization(*data*)
Restore state from the provided data-dict.

text
The raw feature text. Setting this post *Features.TextChanged* and *Features.Changed* notifications.

undo()
Perform an undo if possible, or return. If undo is performed, this will post *BaseObject.BeginUndo* and *BaseObject.EndUndo* notifications.

undoManager
The undo manager assigned to this object.

3.12 Lib

See also:

Notifications: The Lib object uses notifications to notify observers of changes.

3.12.1 Tasks

Notifications

- *dispatcher*
- *addObserver()*
- *removeObserver()*
- *hasObserver()*

Parent

- `getParent()`
- `setParent()`

Lib

class defcon.**Lib**(*font=None*, *layer=None*, *glyph=None*)

This object contains arbitrary data.

This object posts the following notifications:

- Lib.Changed
- Lib.BeginUndo
- Lib.EndUndo
- Lib.BeginRedo
- Lib.EndRedo
- Lib.ItemSet
- Lib.ItemDeleted
- Lib.Cleared
- Lib.Updated

This object behaves like a dict. For example, to get a particular item from the lib:

```
data = lib["com.typesupply.someApplication.blah"]
```

To set the glyph list for a particular group name:

```
lib["com.typesupply.someApplication.blah"] = 123
```

And so on.

Note 1: It is best to keep the data below the top level as shallow as possible. Changes below the top level will go unnoticed by the defcon change notification system. These changes will be saved the next time you save the font, however.

Note 2: The keys used for storing data in the lib should follow the reverse domain naming convention detailed in the UFO specification.

addObserver(*observer*, *methodName*, *notification*, *identifier=None*)

Add an observer to this object's notification dispatcher.

- **observer** An object that can be referenced with weakref.
- **methodName** A string representing the method to be called when the notification is posted.
- **notification** The notification that the observer should be notified of.
- **identifier** None or a string identifying the observation. There is no requirement that the string be unique. A reverse domain naming scheme is recommended, but there are no requirements for the structure of the string.

The method that will be called as a result of the action must accept a single *notification* argument. This will be a `defcon.tools.notifications.Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.addObserver(observer=observer, methodName=methodName,
    notification=notification, observable=anObject, identifier=identifier)
```

canRedo()
Returns a boolean indicating whether the undo manager is able to perform a redo.

canUndo()
Returns a boolean indicating whether the undo manager is able to perform an undo.

clear() → None. Remove all items from D.

copy() → a shallow copy of D

destroyAllRepresentations(*notification=None*)
Destroy all representations.

destroyRepresentation(*name*, ***kwargs*)
Destroy the stored representation for **name** and ****kwargs**. If no **kwargs** are given, any representation with **name** will be destroyed regardless of the **kwargs** passed when the representation was created.

dirty
The dirty state of the object. True if the object has been changed. False if not. Setting this to True will cause the base changed notification to be posted. The object will automatically maintain this attribute and update it as you change the object.

disableNotifications(*notification=None*, *observer=None*)
Disable this object's notifications until told to resume them.

- **notification** The specific notification to disable. This is optional. If no *notification* is given, all notifications will be disabled.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.disableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

dispatcher
The `defcon.tools.notifications.NotificationCenter` assigned to the parent of this object.

enableNotifications(*notification=None*, *observer=None*)
Enable this object's notifications.

- **notification** The specific notification to enable. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.enableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

findObservations(*observer=None*, *notification=None*, *observable=None*, *identifier=None*)
Find observations of this object matching the given arguments based on the values that were passed during addObserver. A value of None for any of these indicates that all should be considered to match the value. In the case of identifier, strings will be matched using fnmatch.fnmatchcase. The returned value will be a list of dictionaries with this format:

[

{ observer=<…> observable=<…> methodName="…" notification="…" identifier="…"

}

]

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.findObservations(
    observer=observer, observable=anObject,
    notification=notification, identifier=identifier
)
```

font
The *Font* that this object belongs to. This should not be set externally.

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

get()
Return the value for key if key is in the dictionary, else default.

getDataForSerialization(***kwargs*)
Return a dict of data that can be pickled.

getRepresentation(*name*, ***kwargs*)
Get a representation. **name** must be a registered representation name. ****kwargs** will be passed to the appropriate representation factory.

glyph
The *Glyph* that this object belongs to (if it isn't a font or layer lib). This should not be set externally.

hasCachedRepresentation(*name*, ***kwargs*)
Returns a boolean indicating if a representation for **name** and ****kwargs** is cached in the object.

hasObserver(*observer*, *notification*)
Returns a boolean indicating is the **observer** is registered for **notification**.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.hasObserver(observer=observer,
    notification=notification, observable=anObject)
```

holdNotifications(*notification=None*, *note=None*)
Hold this object's notifications until told to release them.

- **notification** The specific notification to hold. This is optional. If no *notification* is given, all notifications will be held.

- **note** An arbitrary string containing information about why the hold has been requested, the requester, etc. This is used for reference only.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.holdNotifications(
    observable=anObject, notification=notification, note=note)
```

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

layer
The *Layer* that this object belongs to (if it isn't a font lib). This should not be set externally.

layerSet
The *LayerSet* that this object belongs to (if it isn't a font lib).

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

postNotification(*notification*, *data=None*)
Post a **notification** through this object's notification dispatcher.

- **notification** The name of the notification.
- **data** Arbitrary data that will be stored in the `Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.postNotification(
    notification=notification, observable=anObject, data=data)
```

redo()
Perform a redo if possible, or return. If redo is performed, this will post *BaseObject.BeginRedo* and *BaseObject.EndRedo* notifications.

releaseHeldNotifications(*notification=None*)
Release this object's held notifications.

- **notification** The specific notification to hold. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.releaseHeldNotifications(
    observable=anObject, notification=notification)
```

removeObserver(*observer*, *notification*)
Remove an observer from this object's notification dispatcher.

- **observer** A registered object.
- **notification** The notification that the observer was registered to be notified of.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.removeObserver(observer=observer,
    notification=notification, observable=anObject)
```

representationKeys()
Get a list of all representation keys that are currently cached.

setDataFromSerialization(*data*)
Restore state from the provided data-dict.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

undo()
Perform an undo if possible, or return. If undo is performed, this will post *BaseObject.BeginUndo* and *BaseObject.EndUndo* notifications.

undoManager
The undo manager assigned to this object.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

3.13 Unicode Data

See also:

Notifications: The UnicodeData object uses notifications to notify observers of changes.

3.13.1 Types of Values

This object works with three types of Unicode values: *real*, *pseudo* and *forced*. A *real* Unicode value is the value assigned in the glyph object. A *pseudo*-Unicode value is an educated guess about what the Unicode value for the glyph could be. This guess is made by splitting the suffix, if one exists, off of the glyph name and then looking up the resulting base in the UnicodeData object. If something is found, the value is the pseudo-Unicode value. A *forced*-Unicode value is a Private Use Area value that is temporaryily mapped to a glyph in the font. These values are stored in the font object only as long as the font is active. They will not be saved into the font. **Note:** Forced-Unicode values are very experimental. They should not be relied upon.

3.13.2 Tasks

Value From Glyph Name

- *unicodeForGlyphName*
- *pseudoUnicodeForGlyphName*
- *forcedUnicodeForGlyphName*

Glyph Name from Value

- *glyphNameForForcedUnicode*
- *glyphNameForUnicode*

Glyph Descriptions

- *blockForGlyphName*
- *categoryForGlyphName*
- *scriptForGlyphName*

Open and Closed Relatives

- *closeRelativeForGlyphName*
- *openRelativeForGlyphName*

Decomposition

- *decompositionBaseForGlyphName*

Sorting Glyphs

- *sortGlyphNames()*

Notifications

- *dispatcher*
- *addObserver()*
- *removeObserver()*
- *hasObserver()*

Parent

- getParent()
- setParent()

UnicodeData

class defcon.**UnicodeData**(*layer=None*)

This object serves Unicode data for the font.

This object posts the following notifications:

- UnicodeData.Changed

This object behaves like a dict. The keys are Unicode values and the values are lists of glyph names associated with that unicode value:

```
{
    65 : ["A"],
    66 : ["B"],
}
```

To get the list of glyph names associated with a particular Unicode value, do this:

```
glyphList = unicodeData[65]
```

The object defines many more convenient ways of interacting with this data.

Warning: Setting data into this object manually is *highly* discouraged. The object automatically keeps itself in sync with the font and the glyphs contained in the font. No manual intervention is required.

addGlyphData(*glyphName*, *values*)
Add the data for the glyph with **glyphName** and the Unicode values **values**.

This should never be called directly.

addObserver(*observer*, *methodName*, *notification*, *identifier=None*)
Add an observer to this object's notification dispatcher.

- **observer** An object that can be referenced with weakref.
- **methodName** A string representing the method to be called when the notification is posted.
- **notification** The notification that the observer should be notified of.
- **identifier** None or a string identifying the observation. There is no requirement that the string be unique. A reverse domain naming scheme is recommended, but there are no requirements for the structure of the string.

The method that will be called as a result of the action must accept a single *notification* argument. This will be a `defcon.tools.notifications.Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.addObserver(observer=observer, methodName=methodName,
    notification=notification, observable=anObject, identifier=identifier)
```

blockForGlyphName(*glyphName*, *allowPseudoUnicode=True*)
Get the block for **glyphName**. If **allowPseudoUnicode** is True, a pseudo-Unicode value will be used if needed. This will return *None* if nothing can be found.

canRedo()
Returns a boolean indicating whether the undo manager is able to perform a redo.

canUndo()
Returns a boolean indicating whether the undo manager is able to perform an undo.

categoryForGlyphName(*glyphName*, *allowPseudoUnicode=True*)
Get the category for **glyphName**. If **allowPseudoUnicode** is True, a pseudo-Unicode value will be used if needed. This will return *None* if nothing can be found.

clear()
Completely remove all stored data.

This should never be called directly.

closeRelativeForGlyphName(*glyphName*, *allowPseudoUnicode=True*)
Get the close relative for **glyphName**. For example, if you request the close relative of the glyph name for the character (, you will be given the glyph name for the character) if it exists in the font. If **allowPseudoUnicode** is True, a pseudo-Unicode value will be used if needed. This will return *None* if nothing can be found.

copy() → a shallow copy of D

decompositionBaseForGlyphName(*glyphName*, *allowPseudoUnicode=True*)
Get the decomposition base for **glyphName**. If **allowPseudoUnicode** is True, a pseudo-Unicode value will be used if needed. This will return *glyphName* if nothing can be found.

destroyAllRepresentations(*notification=None*)
Destroy all representations.

destroyRepresentation(*name*, ***kwargs*)
Destroy the stored representation for **name** and ****kwargs**. If no **kwargs** are given, any representation with **name** will be destroyed regardless of the **kwargs** passed when the representation was created.

dirty
The dirty state of the object. True if the object has been changed. False if not. Setting this to True will cause the base changed notification to be posted. The object will automatically maintain this attribute and update it as you change the object.

disableNotifications(*notification=None*, *observer=None*)
Disable this object's notifications until told to resume them.

- **notification** The specific notification to disable. This is optional. If no *notification* is given, all notifications will be disabled.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.disableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

dispatcher
The `defcon.tools.notifications.NotificationCenter` assigned to the parent of this object.

enableNotifications(*notification=None*, *observer=None*)
Enable this object's notifications.

- **notification** The specific notification to enable. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.enableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

findObservations(*observer=None*, *notification=None*, *observable=None*, *identifier=None*)
Find observations of this object matching the given arguments based on the values that were passed during addObserver. A value of None for any of these indicates that all should be considered to match the value. In the case of identifier, strings will be matched using fnmatch.fnmatchcase. The returned value will be a list of dictionaries with this format:

[

{ observer=<…> observable=<…> methodName="…" notification="…" identifier="…"

}

]

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.findObservations(
    observer=observer, observable=anObject,
    notification=notification, identifier=identifier
)
```

font
The *Font* that this object belongs to.

forcedUnicodeForGlyphName(*glyphName*)
Get the forced-Unicode value for **glyphName**.

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

get()
Return the value for key if key is in the dictionary, else default.

getDataForSerialization(***kwargs*)
Return a dict of data that can be pickled.

getRepresentation(*name*, ***kwargs*)
Get a representation. **name** must be a registered representation name. ****kwargs** will be passed to the appropriate representation factory.

glyphNameForForcedUnicode(*value*)
Get the glyph name assigned to the forced-Unicode specified by **value**.

glyphNameForUnicode(*value*)
Get the first glyph assigned to the Unicode specified as **value**. This will return *None* if no glyph is found.

hasCachedRepresentation(*name*, ***kwargs*)
Returns a boolean indicating if a representation for **name** and ****kwargs** is cached in the object.

hasObserver(*observer*, *notification*)
Returns a boolean indicating is the **observer** is registered for **notification**.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.hasObserver(observer=observer,
    notification=notification, observable=anObject)
```

holdNotifications(*notification=None*, *note=None*)
Hold this object's notifications until told to release them.

- **notification** The specific notification to hold. This is optional. If no *notification* is given, all notifications will be held.
- **note** An arbitrary string containing information about why the hold has been requested, the requester, etc. This is used for reference only.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.holdNotifications(
    observable=anObject, notification=notification, note=note)
```

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

layer
The *Layer* that this object belongs to.

layerSet
The *LayerSet* that this object belongs to.

openRelativeForGlyphName(*glyphName*, *allowPseudoUnicode=True*)
Get the open relative for **glyphName**. For example, if you request the open relative of the glyph name for the character), you will be given the glyph name for the character (if it exists in the font. If **allowPseudoUnicode** is True, a pseudo-Unicode value will be used if needed. This will return *None* if nothing can be found.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

postNotification(*notification*, *data=None*)
Post a **notification** through this object's notification dispatcher.

- **notification** The name of the notification.
- **data** Arbitrary data that will be stored in the `Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.postNotification(
    notification=notification, observable=anObject, data=data)
```

pseudoUnicodeForGlyphName(*glyphName*)
Get the pseudo-Unicode value for **glyphName**. This will return *None* if nothing is found.

redo()
Perform a redo if possible, or return. If redo is performed, this will post *BaseObject.BeginRedo* and *BaseObject.EndRedo* notifications.

releaseHeldNotifications(*notification=None*)
Release this object's held notifications.

- **notification** The specific notification to hold. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.releaseHeldNotifications(
    observable=anObject, notification=notification)
```

removeGlyphData(*glyphName*, *values*)
Remove the data for the glyph with **glyphName** and the Unicode values **values**.

This should never be called directly.

removeObserver(*observer*, *notification*)
Remove an observer from this object's notification dispatcher.

- **observer** A registered object.
- **notification** The notification that the observer was registered to be notified of.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.removeObserver(observer=observer,
    notification=notification, observable=anObject)
```

representationKeys()
Get a list of all representation keys that are currently cached.

scriptForGlyphName(*glyphName*, *allowPseudoUnicode=True*)
Get the script for **glyphName**. If **allowPseudoUnicode** is True, a pseudo-Unicode value will be used if needed. This will return *None* if nothing can be found.

setDataFromSerialization(*data*)
Restore state from the provided data-dict.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

sortGlyphNames(*glyphNames, sortDescriptors=[{'type': 'unicode'}]*)
This sorts the list of **glyphNames** following the sort descriptors provided in the **sortDescriptors** list. This works by iterating over the sort descriptors and subdividing. For example, if the first sort descriptor is a suffix type, internally, the result of the sort will look something like this:

```
[
    [glyphsWithNoSuffix],
    [glyphsWith.suffix1],
    [glyphsWith.suffix2]
]
```

When the second sort descriptor is processed, the results of previous sorts are subdivided even further. For example, if the second sort type is script:

```
[[
    [glyphsWithNoSuffix, script1], [glyphsWithNoSuffix, script2],
    [glyphsWith.suffix1, script1], [glyphsWith.suffix1, script2],
    [glyphsWith.suffix2, script1], [glyphsWith.suffix2, script2]
]]
```

And so on. The returned list will be flattened into a list of glyph names.

Each item in **sortDescriptors** should be a dict of the following form:

Key	Description
type	The type of sort to perform. See below for options.
ascending	Boolean representing if the glyphs should be in ascending or descending order. Optional. The default is True.
allowPseudoUnicode	Boolean representing if pseudo-Unicode values are used. If not, real Unicode values will be used if necessary. Optional. The default is False.
function	A function. Used only for **custom** sort types. See details below.

Available Sort Types:

There are four types of sort types: simple, complex, canned and custom. Simple sorts are based on sorting non-magical values, such as Unicode values. Complex sorts are heuristic based sorts based on common glyph name practices, aesthetic preferences and other hard to quantify ideas. Custom sorts are just that, custom sorts. Canned sorts are combinations of simple, complex and custom sorts that give optimized ordering results. Complex and canned sorts may change with further updates, so they should not be relied on for persistent ordering.

Simple Sort Types	Description
alphabetical	Self-explanatory.
unicode	Sort based on Unicode value.
script	Sort based on Unicode script.
category	Sort based on Unicode category.
block	Sort based on Unicode block.
suffix	Sort based on glyph name suffix.
decompositionBase	Sort based on the base glyph defined in the decomposition rules.

Complex Sort Types	Description
weightedSuffix	Sort based on glyph names suffix. The ordering of the suffixes is based on the calculated "weight" of the suffix. This value is calculated by noting what type of glyphs have the same suffix. The more glyph types, the more important the suffix. Additionally, glyphs with suffixes that have only numerical differences are grouped together. For example, a.alt, a.alt1 and a.alt319 will be grouped together.
ligature	Sort into to groups: non-ligatures and ligatures. The determination of whether a glyph is a ligature or not is based on the Unicode value, common glyph names or the use of an underscore in the name.

Canned Sort Types	Description
cannedDesign	Sort glyphs into a design process friendly order.

Custom Sort Type	Description
custom	Sort using a custom function. See details below.

Sorting with a custom function: If the builtin sort types don't do exactly what you need, you can use a **custom** sort type that contains an arbitrary function that handles sorting externally. This follows the same sorting logic as detailed above. The custom sort type can be used in conjunction with the builtin sort types.

The function should follow this form:

```
mySortFunction(font, glyphNames, ascending=True, allowPseudoUnicode=False)
```

The **ascending** and **allowPseudoUnicode** arguments will be the values defined in the sort descriptors.

The function should return a list of lists of glyph names.

An example:

```
def sortByE(font, glyphNames, ascending=True, allowsPseudoUnicodes=False):
    startsWithE = []
    doesNotStartWithE = []
    for glyphName in glyphNames:
        if glyphName.startswith("startsWithE"):
            startsWithE.append(glyphName)
```

(continues on next page)

(continued from previous page)

```
        else:
            doesNotStartWithE.append(glyphName)
    return [startsWithE, doesNotStartWithE]
```

undo()
Perform an undo if possible, or return. If undo is performed, this will post *BaseObject.BeginUndo* and *BaseObject.EndUndo* notifications.

undoManager
The undo manager assigned to this object.

unicodeForGlyphName(*glyphName*)
Get the Unicode value for **glyphName**. Returns *None* if no value is found.

update(*other*)
Update the data int this object with the data from **other**.

This should never be called directly.

values() → an object providing a view on D's values

3.14 NotificationCenter

Direct creation of and interation with these objects will most likely be rare as they are automatically handled by `BaseObject`.

3.14.1 NotificationCenter

class `defcon.tools.notifications.`**NotificationCenter**

addObserver(*observer*, *methodName*, *notification=None*, *observable=None*, *identifier=None*)
Add an observer to this notification dispatcher.

- **observer** An object that can be referenced with weakref.
- **methodName** A string representing the method to be called when the notification is posted.
- **notification** The notification that the observer should be notified of. If this is None, all notifications for the *observable* will be posted to *observer*.
- **observable** The object to observe. If this is None, all notifications with the name provided as *notification* will be posted to the *observer*.
- **identifier** None or a string identifying the observation. There is no requirement that the string be unique. A reverse domain naming scheme is recommended, but there are no requirements for the structure of the string.

If None is given for both *notification* and *observable* **all** notifications posted will be sent to the given method of the observer.

The method that will be called as a result of the action must accept a single *notification* argument. This will be a `Notification` object.

areNotificationsDisabled(*observable=None*, *notification=None*, *observer=None*)
Returns a boolean indicating if notifications posted to all objects observing **notification** in **observable** are disabled.

- **observable** The object that the notification belongs to. This is optional.
- **notification** The name of the notification. This is optional.
- **observer** The observer. This is optional.

areNotificationsHeld(*observable=None*, *notification=None*, *observer=None*)
Returns a boolean indicating if notifications posted to all objects observing **notification** in **observable** are being held.

- **observable** The object that the notification belongs to. This is optional.
- **notification** The name of the notification. This is optional.
- **observer** The observer. This is optional.

disableNotifications(*observable=None*, *notification=None*, *observer=None*)
Disable all posts of **notification** from **observable** posted to **observer** observing.

- **observable** The object that the notification belongs to. This is optional. If no *observable* is given, *all notifications* will be disabled for *observer*.
- **notification** The name of the notification. This is optional. If no *notification* is given, *all* notifications for *observable* will be disabled for *observer*.
- **observer** The specific observer to not send posts to. If no *observer* is given, the appropriate notifications will not be posted to any observers.

This object will retain a count of how many times it has been told to disable notifications for *notification* and *observable*. It will not enable new notifications until the *notification* and *observable* have been released the same number of times.

enableNotifications(*observable=None*, *notification=None*, *observer=None*)
Enable notifications posted to all objects observing **notification** in **observable**.

- **observable** The object that the notification belongs to. This is optional.
- **notification** The name of the notification. This is optional.
- **observer** The observer. This is optional.

findObservations(*observer=None*, *notification=None*, *observable=None*, *identifier=None*)
Find observations matching the given arguments based on the values that were passed during addObserver. A value of None for any of these indicates that all should be considered to match the value. In the case of identifier, strings will be matched using fnmatch.fnmatchcase. The returned value will be a list of dictionaries with this format:

[

{ observer=<...> observable=<...> methodName="..." notification="..." identifier="..."

}

]

getHeldNotificationNotes(*observable=None*, *notification=None*, *observer=None*)
Returns a list of notes defined for notification holds observing **notification** in **observable** are being held.

- **observable** The object that the notification belongs to. This is optional.
- **notification** The name of the notification. This is optional.
- **observer** The observer. This is optional.

getHeldNotifications()
Returns a list of all held notifications. This will be a tuple of the form:

(notification, observable, observer)

hasObserver(*observer*, *notification*, *observable*)
Returns a boolean indicating if the **observer** is registered for **notification** posted by **observable**. Either *observable* or *notification* may be None.

holdNotifications(*observable=None*, *notification=None*, *observer=None*, *note=None*)
Hold all notifications posted to all objects observing **notification** in **observable**.

- **observable** The object that the notification belongs to. This is optional. If no *observable* is given, *all notifications* will be held.
- **notification** The name of the notification. This is optional. If no *notification* is given, *all* notifications for *observable* will be held.
- **observer** The specific observer to not hold notifications for. If no *observer* is given, the appropriate notifications will be held for all observers.
- **note** An arbitrary string containing information about why the hold has been requested, the requester, etc. This is used for reference only.

Held notifications will be posted after the matching *notification* and *observable* have been passed to `Notification.releaseHeldNotifications()`. This object will retain a count of how many times it has been told to hold notifications for *notification* and *observable*. It will not post the notifications until the *notification* and *observable* have been released the same number of times.

releaseHeldNotifications(*observable=None*, *notification=None*, *observer=None*)
Release all held notifications posted to all objects observing **notification** in **observable**.

- **observable** The object that the notification belongs to. This is optional.
- **notification** The name of the notification. This is optional.
- **observer** The observer. This is optional.

removeObserver(*observer*, *notification*, *observable=None*)
Remove an observer from this notification dispatcher.

- **observer** A registered object.
- **notification** The notification that the observer was registered to be notified of. If this is *"all"*, all notifications for the *observable* will be removed for *observer*.
- **observable** The object being observed.

3.14.2 Notification

class `defcon.tools.notifications.`**Notification**(*name*, *objRef*, *data*)
An object that wraps notification data.

data
Arbitrary data passed along with the notification. There is no set format for this data and there is not requirement that any data be present. Refer to the documentation for methods that are responsible for generating notifications for information about this data.

name
The notification name. A string.

object
The observable object the notification belongs to.

3.15 BaseObject

The main objects in defcon all subclass these objects.

See also:

`NotificationCenter` The base object uses notifications to notify observers about changes. The API for subscribing/unsubscribing to notifications are detailed below. Some familiarity with the `NotificationCenter` might be helpful.

3.15.1 BaseObject

class `defcon.objects.base.`**`BaseObject`**

The base object in defcon from which all other objects should be derived.

This object posts the following notifications:

- BaseObject.Changed
- BaseObject.BeginUndo
- BaseObject.EndUndo
- BaseObject.BeginRedo
- BaseObject.EndRedo

Keep in mind that subclasses will not post these same notifications.

Subclasses must override the following attributes:

Name	Notes
changeNotificationName	This must be a string unique to the class indicating the name of the notification to be posted when the dirty attribute is set.
representationFactories	This must be a dictionary that is shared across *all* instances of the class.

`addObserver`(*observer*, *methodName*, *notification*, *identifier=None*)

Add an observer to this object's notification dispatcher.

- **observer** An object that can be referenced with weakref.
- **methodName** A string representing the method to be called when the notification is posted.
- **notification** The notification that the observer should be notified of.
- **identifier** None or a string identifying the observation. There is no requirement that the string be unique. A reverse domain naming scheme is recommended, but there are no requirements for the structure of the string.

The method that will be called as a result of the action must accept a single *notification* argument. This will be a *`defcon.tools.notifications.Notification`* object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.addObserver(observer=observer, methodName=methodName,
    notification=notification, observable=anObject, identifier=identifier)
```

canRedo()
Returns a boolean indicating whether the undo manager is able to perform a redo.

canUndo()
Returns a boolean indicating whether the undo manager is able to perform an undo.

destroyAllRepresentations(*notification=None*)
Destroy all representations.

destroyRepresentation(*name*, ***kwargs*)
Destroy the stored representation for **name** and ****kwargs**. If no **kwargs** are given, any representation with **name** will be destroyed regardless of the **kwargs** passed when the representation was created.

dirty
The dirty state of the object. True if the object has been changed. False if not. Setting this to True will cause the base changed notification to be posted. The object will automatically maintain this attribute and update it as you change the object.

disableNotifications(*notification=None*, *observer=None*)
Disable this object's notifications until told to resume them.

- **notification** The specific notification to disable. This is optional. If no *notification* is given, all notifications will be disabled.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.disableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

dispatcher
The *defcon.tools.notifications.NotificationCenter* assigned to the parent of this object.

enableNotifications(*notification=None*, *observer=None*)
Enable this object's notifications.

- **notification** The specific notification to enable. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.enableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

findObservations(*observer=None*, *notification=None*, *observable=None*, *identifier=None*)
Find observations of this object matching the given arguments based on the values that were passed during addObserver. A value of None for any of these indicates that all should be considered to match the value. In the case of identifier, strings will be matched using fnmatch.fnmatchcase. The returned value will be a list of dictionaries with this format:

[

{ observer=<…> observable=<…> methodName="…" notification="…" identifier="…"

}

]

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.findObservations(
    observer=observer, observable=anObject,
    notification=notification, identifier=identifier
)
```

getDataForSerialization(***kwargs*)
Return a dict of data that can be pickled.

getRepresentation(*name*, ***kwargs*)
Get a representation. **name** must be a registered representation name. ****kwargs** will be passed to the appropriate representation factory.

hasCachedRepresentation(*name*, ***kwargs*)
Returns a boolean indicating if a representation for **name** and ****kwargs** is cached in the object.

hasObserver(*observer*, *notification*)
Returns a boolean indicating is the **observer** is registered for **notification**.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.hasObserver(observer=observer,
    notification=notification, observable=anObject)
```

holdNotifications(*notification=None*, *note=None*)
Hold this object's notifications until told to release them.

- **notification** The specific notification to hold. This is optional. If no *notification* is given, all notifications will be held.
- **note** An arbitrary string containing information about why the hold has been requested, the requester, etc. This is used for reference only.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.holdNotifications(
    observable=anObject, notification=notification, note=note)
```

postNotification(*notification*, *data=None*)
Post a **notification** through this object's notification dispatcher.

- **notification** The name of the notification.
- **data** Arbitrary data that will be stored in the `Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.postNotification(
    notification=notification, observable=anObject, data=data)
```

redo()
Perform a redo if possible, or return. If redo is performed, this will post *BaseObject.BeginRedo* and *BaseObject.EndRedo* notifications.

releaseHeldNotifications(*notification=None*)
Release this object's held notifications.

- **notification** The specific notification to hold. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.releaseHeldNotifications(
    observable=anObject, notification=notification)
```

removeObserver(*observer*, *notification*)
Remove an observer from this object's notification dispatcher.

- **observer** A registered object.
- **notification** The notification that the observer was registered to be notified of.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.removeObserver(observer=observer,
    notification=notification, observable=anObject)
```

representationKeys()
Get a list of all representation keys that are currently cached.

setDataFromSerialization(*data*)
Restore state from the provided data-dict.

undo()
Perform an undo if possible, or return. If undo is performed, this will post *BaseObject.BeginUndo* and *BaseObject.EndUndo* notifications.

undoManager
The undo manager assigned to this object.

3.15.2 BaseDictObject

class defcon.objects.base.**BaseDictObject**
A subclass of BaseObject that implements a dict API. Any changes to the contents of the object will cause the dirty attribute to be set to True.

3.16 LayerSet

3.16.1 LayerSet

class defcon.**LayerSet**(*font=None*, *layerClass=None*, *libClass=None*, *unicodeDataClass=None*, *guidelineClass=None*, *glyphClass=None*, *glyphContourClass=None*, *glyphPointClass=None*, *glyphComponentClass=None*, *glyphAnchorClass=None*, *glyphImageClass=None*)
This object manages all layers in the font.

This object posts the following notifications:

- LayerSet.Changed
- LayerSet.LayersChanged
- LayerSet.LayerChanged
- LayerSet.DefaultLayerWillChange

- LayerSet.DefaultLayerChanged
- LayerSet.LayerOrderChanged
- LayerSet.LayerAdded
- LayerSet.LayerDeleted
- LayerSet.LayerWillBeDeleted
- LayerSet.LayerNameChanged

This object behaves like a dict. For example, to get a particular layer:

```
layer = layerSet["layer name"]
```

If the layer name is None, the default layer will be retrieved.

Note: It's up to the caller to ensure that a default layer is present as required by the UFO specification.

addObserver(*observer*, *methodName*, *notification*, *identifier=None*)
Add an observer to this object's notification dispatcher.

- **observer** An object that can be referenced with weakref.
- **methodName** A string representing the method to be called when the notification is posted.
- **notification** The notification that the observer should be notified of.
- **identifier** None or a string identifying the observation. There is no requirement that the string be unique. A reverse domain naming scheme is recommended, but there are no requirements for the structure of the string.

The method that will be called as a result of the action must accept a single *notification* argument. This will be a `defcon.tools.notifications.Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.addObserver(observer=observer, methodName=methodName,
    notification=notification, observable=anObject, identifier=identifier)
```

canRedo()
Returns a boolean indicating whether the undo manager is able to perform a redo.

canUndo()
Returns a boolean indicating whether the undo manager is able to perform an undo.

defaultLayer
The default `Layer` object. Setting this will post *LayerSet.DefaultLayerChanged* and *LayerSet.Changed* notifications.

destroyAllRepresentations(*notification=None*)
Destroy all representations.

destroyRepresentation(*name*, ***kwargs*)
Destroy the stored representation for **name** and ****kwargs**. If no **kwargs** are given, any representation with **name** will be destroyed regardless of the **kwargs** passed when the representation was created.

dirty
The dirty state of the object. True if the object has been changed. False if not. Setting this to True will cause the base changed notification to be posted. The object will automatically maintain this attribute and update it as you change the object.

disableNotifications(*notification=None*, *observer=None*)
Disable this object's notifications until told to resume them.

- **notification** The specific notification to disable. This is optional. If no *notification* is given, all notifications will be disabled.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.disableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

dispatcher
The `defcon.tools.notifications.NotificationCenter` assigned to the parent of this object.

enableNotifications(*notification=None*, *observer=None*)
Enable this object's notifications.

- **notification** The specific notification to enable. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.enableNotifications(
    observable=anObject, notification=notification, observer=observer)
```

findObservations(*observer=None*, *notification=None*, *observable=None*, *identifier=None*)
Find observations of this object matching the given arguments based on the values that were passed during addObserver. A value of None for any of these indicates that all should be considered to match the value. In the case of identifier, strings will be matched using fnmatch.fnmatchcase. The returned value will be a list of dictionaries with this format:

[

{ observer=<...> observable=<...> methodName="..." notification="..." identifier="..."

}

]

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.findObservations(
    observer=observer, observable=anObject,
    notification=notification, identifier=identifier
)
```

font
The `Font` that this layer set belongs to.

getDataForSerialization(***kwargs*)
Return a dict of data that can be pickled.

getRepresentation(*name*, ***kwargs*)
Get a representation. **name** must be a registered representation name. ****kwargs** will be passed to the appropriate representation factory.

getSaveProgressBarTickCount(*formatVersion*)
Get the number of ticks that will be used by a progress bar in the save method. This method should not be called externally. Subclasses may override this method to implement custom saving behavior.

hasCachedRepresentation(*name*, ***kwargs*)
Returns a boolean indicating if a representation for **name** and ****kwargs** is cached in the object.

hasObserver(*observer*, *notification*)
Returns a boolean indicating is the **observer** is registered for **notification**.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.hasObserver(observer=observer,
    notification=notification, observable=anObject)
```

holdNotifications(*notification=None*, *note=None*)
Hold this object's notifications until told to release them.

- **notification** The specific notification to hold. This is optional. If no *notification* is given, all notifications will be held.
- **note** An arbitrary string containing information about why the hold has been requested, the requester, etc. This is used for reference only.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.holdNotifications(
    observable=anObject, notification=notification, note=note)
```

layerOrder
The layer order from top to bottom. Setting this will post *LayerSet.LayerOrderChanged* and *LayerSet.Changed* notifications.

newLayer(*name*, *glyphSet=None*)
Create a new `Layer` and add it to the top of the layer order. **glyphSet** should only be passed when reading from a UFO.

This posts *LayerSet.LayerAdded* and *LayerSet.Changed* notifications.

postNotification(*notification*, *data=None*)
Post a **notification** through this object's notification dispatcher.

- **notification** The name of the notification.
- **data** Arbitrary data that will be stored in the `Notification` object.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.postNotification(
    notification=notification, observable=anObject, data=data)
```

redo()
Perform a redo if possible, or return. If redo is performed, this will post *BaseObject.BeginRedo* and *BaseObject.EndRedo* notifications.

releaseHeldNotifications(*notification=None*)
Release this object's held notifications.

- **notification** The specific notification to hold. This is optional.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.releaseHeldNotifications(
    observable=anObject, notification=notification)
```

reloadLayers(*layerData*)
Reload the layers. This should not be called externally.

removeObserver(*observer*, *notification*)
Remove an observer from this object's notification dispatcher.

- **observer** A registered object.
- **notification** The notification that the observer was registered to be notified of.

This is a convenience method that does the same thing as:

```
dispatcher = anObject.dispatcher
dispatcher.removeObserver(observer=observer,
    notification=notification, observable=anObject)
```

representationKeys()
Get a list of all representation keys that are currently cached.

save(*writer*, *saveAs=False*, *progressBar=None*)
Save all layers. This method should not be called externally. Subclasses may override this method to implement custom saving behavior.

setDataFromSerialization(*data*)
Restore state from the provided data-dict.

testForExternalChanges(*reader*)
Test for external changes. This should not be called externally.

undo()
Perform an undo if possible, or return. If undo is performed, this will post *BaseObject.BeginUndo* and *BaseObject.EndUndo* notifications.

undoManager
The undo manager assigned to this object.

CHAPTER 4

Dependencies

- FontTools >= 3.31.0, installed with the *fonttools[ufo]* extra, required to import `fonttools.ufoLib` module.

CHAPTER 5

Optional Dependencies

- fontPens
- lxml

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

defcon, 78
defcon.objects.base, 88
defcon.tools.notifications, 85

Index

A

addGlyphData() (*defcon.UnicodeData method*), 79
addObserver() (*defcon.Anchor method*), 48
addObserver() (*defcon.Component method*), 42
addObserver() (*defcon.Contour method*), 36
addObserver() (*defcon.Features method*), 69
addObserver() (*defcon.Font method*), 11
addObserver() (*defcon.Glyph method*), 27
addObserver() (*defcon.Groups method*), 65
addObserver() (*defcon.Info method*), 56
addObserver() (*defcon.Kerning method*), 60
addObserver() (*defcon.Layer method*), 19
addObserver() (*defcon.LayerSet method*), 92
addObserver() (*defcon.Lib method*), 73
addObserver() (*defcon.objects.base.BaseObject method*), 88
addObserver() (*defcon.tools.notifications.NotificationCenter method*), 85
addObserver() (*defcon.UnicodeData method*), 79
addPoint() (*defcon.Contour method*), 37
Anchor (*class in defcon*), 48
anchorClass (*defcon.Glyph attribute*), 28
anchorIndex() (*defcon.Glyph method*), 28
anchors (*defcon.Glyph attribute*), 28
appendAnchor() (*defcon.Glyph method*), 28
appendComponent() (*defcon.Glyph method*), 28
appendContour() (*defcon.Glyph method*), 28
appendGuideline() (*defcon.Font method*), 12
appendGuideline() (*defcon.Glyph method*), 28
appendPoint() (*defcon.Contour method*), 37
area (*defcon.Contour attribute*), 37
area (*defcon.Glyph attribute*), 28
areNotificationsDisabled() (*defcon.tools.notifications.NotificationCenter method*), 85
areNotificationsHeld() (*defcon.tools.notifications.NotificationCenter method*), 86

B

BaseDictObject (*class in defcon.objects.base*), 91
baseGlyph (*defcon.Component attribute*), 43
BaseObject (*class in defcon.objects.base*), 88
beginPath() (*defcon.Contour method*), 37
blockForGlyphName() (*defcon.UnicodeData method*), 79
bottomMargin (*defcon.Glyph attribute*), 28
bounds (*defcon.Component attribute*), 43
bounds (*defcon.Contour attribute*), 37
bounds (*defcon.Font attribute*), 12
bounds (*defcon.Glyph attribute*), 28
bounds (*defcon.Layer attribute*), 20

C

canRedo() (*defcon.Anchor method*), 49
canRedo() (*defcon.Component method*), 43
canRedo() (*defcon.Contour method*), 37
canRedo() (*defcon.Features method*), 70
canRedo() (*defcon.Font method*), 12
canRedo() (*defcon.Glyph method*), 28
canRedo() (*defcon.Groups method*), 65
canRedo() (*defcon.Info method*), 56
canRedo() (*defcon.Kerning method*), 60
canRedo() (*defcon.Layer method*), 20
canRedo() (*defcon.LayerSet method*), 92
canRedo() (*defcon.Lib method*), 74
canRedo() (*defcon.objects.base.BaseObject method*), 88
canRedo() (*defcon.UnicodeData method*), 79
canUndo() (*defcon.Anchor method*), 49
canUndo() (*defcon.Component method*), 43
canUndo() (*defcon.Contour method*), 37
canUndo() (*defcon.Features method*), 70
canUndo() (*defcon.Font method*), 12
canUndo() (*defcon.Glyph method*), 28
canUndo() (*defcon.Groups method*), 66
canUndo() (*defcon.Info method*), 56
canUndo() (*defcon.Kerning method*), 60

canUndo() (*defcon.Layer method*), 20
canUndo() (*defcon.LayerSet method*), 92
canUndo() (*defcon.Lib method*), 74
canUndo() (*defcon.objects.base.BaseObject method*), 89
canUndo() (*defcon.UnicodeData method*), 79
categoryForGlyphName() (*defcon.UnicodeData method*), 79
clear() (*defcon.Anchor method*), 49
clear() (*defcon.Contour method*), 37
clear() (*defcon.Glyph method*), 28
clear() (*defcon.Groups method*), 66
clear() (*defcon.Kerning method*), 61
clear() (*defcon.Lib method*), 74
clear() (*defcon.UnicodeData method*), 79
clearAnchors() (*defcon.Glyph method*), 28
clearComponents() (*defcon.Glyph method*), 29
clearContours() (*defcon.Glyph method*), 29
clearGuidelines() (*defcon.Font method*), 12
clearGuidelines() (*defcon.Glyph method*), 29
clockwise (*defcon.Contour attribute*), 37
closeRelativeForGlyphName() (*defcon.UnicodeData method*), 80
color (*defcon.Anchor attribute*), 49
color (*defcon.Layer attribute*), 20
Component (*class in defcon*), 42
componentClass (*defcon.Glyph attribute*), 29
componentIndex() (*defcon.Glyph method*), 29
componentReferences (*defcon.Font attribute*), 12
componentReferences (*defcon.Layer attribute*), 20
components (*defcon.Glyph attribute*), 29
Contour (*class in defcon*), 36
contourClass (*defcon.Glyph attribute*), 29
contourIndex() (*defcon.Glyph method*), 29
contourInside() (*defcon.Contour method*), 37
controlPointBounds (*defcon.Component attribute*), 43
controlPointBounds (*defcon.Contour attribute*), 37
controlPointBounds (*defcon.Font attribute*), 12
controlPointBounds (*defcon.Glyph attribute*), 29
controlPointBounds (*defcon.Layer attribute*), 20
copy() (*defcon.Anchor method*), 49
copy() (*defcon.Groups method*), 66
copy() (*defcon.Kerning method*), 61
copy() (*defcon.Lib method*), 74
copy() (*defcon.UnicodeData method*), 80
copyDataFromGlyph() (*defcon.Glyph method*), 29
correctContourDirection() (*defcon.Glyph method*), 29

D

data (*defcon.Font attribute*), 12
data (*defcon.tools.notifications.Notification attribute*), 87
decomposeAllComponents() (*defcon.Glyph method*), 29
decomposeComponent() (*defcon.Glyph method*), 29
decompositionBaseForGlyphName() (*defcon.UnicodeData method*), 80
defaultLayer (*defcon.LayerSet attribute*), 92
defcon (*module*), 11, 19, 26, 36, 42, 47, 48, 56, 59, 64, 69, 73, 78, 91
defcon.objects.base (*module*), 88
defcon.tools.notifications (*module*), 85
destroyAllRepresentations() (*defcon.Anchor method*), 49
destroyAllRepresentations() (*defcon.Component method*), 43
destroyAllRepresentations() (*defcon.Contour method*), 37
destroyAllRepresentations() (*defcon.Features method*), 70
destroyAllRepresentations() (*defcon.Font method*), 12
destroyAllRepresentations() (*defcon.Glyph method*), 29
destroyAllRepresentations() (*defcon.Groups method*), 66
destroyAllRepresentations() (*defcon.Info method*), 56
destroyAllRepresentations() (*defcon.Kerning method*), 61
destroyAllRepresentations() (*defcon.Layer method*), 20
destroyAllRepresentations() (*defcon.LayerSet method*), 92
destroyAllRepresentations() (*defcon.Lib method*), 74
destroyAllRepresentations() (*defcon.objects.base.BaseObject method*), 89
destroyAllRepresentations() (*defcon.UnicodeData method*), 80
destroyRepresentation() (*defcon.Anchor method*), 49
destroyRepresentation() (*defcon.Component method*), 43
destroyRepresentation() (*defcon.Contour method*), 37
destroyRepresentation() (*defcon.Features method*), 70
destroyRepresentation() (*defcon.Font method*), 12
destroyRepresentation() (*defcon.Glyph method*), 29
destroyRepresentation() (*defcon.Groups method*), 66
destroyRepresentation() (*defcon.Info method*), 56

destroyRepresentation() (*defcon.Kerning method*), 61
destroyRepresentation() (*defcon.Layer method*), 20
destroyRepresentation() (*defcon.LayerSet method*), 92
destroyRepresentation() (*defcon.Lib method*), 74
destroyRepresentation() (*defcon.objects.base.BaseObject method*), 89
destroyRepresentation() (*defcon.UnicodeData method*), 80
dirty (*defcon.Anchor attribute*), 49
dirty (*defcon.Component attribute*), 43
dirty (*defcon.Contour attribute*), 37
dirty (*defcon.Features attribute*), 70
dirty (*defcon.Font attribute*), 12
dirty (*defcon.Glyph attribute*), 30
dirty (*defcon.Groups attribute*), 66
dirty (*defcon.Info attribute*), 56
dirty (*defcon.Kerning attribute*), 61
dirty (*defcon.Layer attribute*), 20
dirty (*defcon.LayerSet attribute*), 92
dirty (*defcon.Lib attribute*), 74
dirty (*defcon.objects.base.BaseObject attribute*), 89
dirty (*defcon.UnicodeData attribute*), 80
disableNotifications() (*defcon.Anchor method*), 49
disableNotifications() (*defcon.Component method*), 43
disableNotifications() (*defcon.Contour method*), 38
disableNotifications() (*defcon.Features method*), 70
disableNotifications() (*defcon.Font method*), 12
disableNotifications() (*defcon.Glyph method*), 30
disableNotifications() (*defcon.Groups method*), 66
disableNotifications() (*defcon.Info method*), 57
disableNotifications() (*defcon.Kerning method*), 61
disableNotifications() (*defcon.Layer method*), 20
disableNotifications() (*defcon.LayerSet method*), 92
disableNotifications() (*defcon.Lib method*), 74
disableNotifications() (*defcon.objects.base.BaseObject method*), 89
disableNotifications() (*defcon.tools.notifications.NotificationCenter method*), 86
disableNotifications() (*defcon.UnicodeData method*), 80
dispatcher (*defcon.Anchor attribute*), 49
dispatcher (*defcon.Component attribute*), 43
dispatcher (*defcon.Contour attribute*), 38
dispatcher (*defcon.Features attribute*), 70
dispatcher (*defcon.Font attribute*), 13
dispatcher (*defcon.Glyph attribute*), 30
dispatcher (*defcon.Groups attribute*), 66
dispatcher (*defcon.Info attribute*), 57
dispatcher (*defcon.Kerning attribute*), 61
dispatcher (*defcon.Layer attribute*), 20
dispatcher (*defcon.LayerSet attribute*), 93
dispatcher (*defcon.Lib attribute*), 74
dispatcher (*defcon.objects.base.BaseObject attribute*), 89
dispatcher (*defcon.UnicodeData attribute*), 80
draw() (*defcon.Component method*), 44
draw() (*defcon.Contour method*), 38
draw() (*defcon.Glyph method*), 30
drawPoints() (*defcon.Component method*), 44
drawPoints() (*defcon.Contour method*), 38
drawPoints() (*defcon.Glyph method*), 30

E

enableNotifications() (*defcon.Anchor method*), 49
enableNotifications() (*defcon.Component method*), 44
enableNotifications() (*defcon.Contour method*), 38
enableNotifications() (*defcon.Features method*), 70
enableNotifications() (*defcon.Font method*), 13
enableNotifications() (*defcon.Glyph method*), 30
enableNotifications() (*defcon.Groups method*), 66
enableNotifications() (*defcon.Info method*), 57
enableNotifications() (*defcon.Kerning method*), 61
enableNotifications() (*defcon.Layer method*), 21
enableNotifications() (*defcon.LayerSet method*), 93
enableNotifications() (*defcon.Lib method*), 74
enableNotifications() (*defcon.objects.base.BaseObject method*), 89
enableNotifications() (*defcon.tools.notifications.NotificationCenter method*), 86
enableNotifications() (*defcon.UnicodeData method*), 80
endPath() (*defcon.Contour method*), 38

F

Features (*class in defcon*), 69
features (*defcon.Font attribute*), 13
find() (*defcon.Kerning method*), 61
findObservations() (*defcon.Anchor method*), 49
findObservations() (*defcon.Component method*), 44
findObservations() (*defcon.Contour method*), 38
findObservations() (*defcon.Features method*), 70
findObservations() (*defcon.Font method*), 13
findObservations() (*defcon.Glyph method*), 30
findObservations() (*defcon.Groups method*), 66
findObservations() (*defcon.Info method*), 57
findObservations() (*defcon.Kerning method*), 61
findObservations() (*defcon.Layer method*), 21
findObservations() (*defcon.LayerSet method*), 93
findObservations() (*defcon.Lib method*), 75
findObservations() (*defcon.objects.base.BaseObject method*), 89
findObservations() (*defcon.tools.notifications.NotificationCenter method*), 86
findObservations() (*defcon.UnicodeData method*), 80
Font (*class in defcon*), 11
font (*defcon.Anchor attribute*), 50
font (*defcon.Component attribute*), 44
font (*defcon.Contour attribute*), 38
font (*defcon.Features attribute*), 71
font (*defcon.Glyph attribute*), 31
font (*defcon.Groups attribute*), 67
font (*defcon.Info attribute*), 57
font (*defcon.Kerning attribute*), 62
font (*defcon.Layer attribute*), 21
font (*defcon.LayerSet attribute*), 93
font (*defcon.Lib attribute*), 75
font (*defcon.UnicodeData attribute*), 81
forcedUnicodeForGlyphName() (*defcon.UnicodeData method*), 81
fromkeys() (*defcon.Anchor method*), 50
fromkeys() (*defcon.Groups method*), 67
fromkeys() (*defcon.Kerning method*), 62
fromkeys() (*defcon.Lib method*), 75
fromkeys() (*defcon.UnicodeData method*), 81

G

generateIdentifier() (*defcon.Anchor method*), 50
generateIdentifier() (*defcon.Component method*), 44
generateIdentifier() (*defcon.Contour method*), 39
generateIdentifierForPoint() (*defcon.Contour method*), 39
get() (*defcon.Anchor method*), 50
get() (*defcon.Groups method*), 67
get() (*defcon.Kerning method*), 62
get() (*defcon.Lib method*), 75
get() (*defcon.UnicodeData method*), 81
getDataForSerialization() (*defcon.Anchor method*), 50
getDataForSerialization() (*defcon.Component method*), 44
getDataForSerialization() (*defcon.Contour method*), 39
getDataForSerialization() (*defcon.Features method*), 71
getDataForSerialization() (*defcon.Font method*), 13
getDataForSerialization() (*defcon.Glyph method*), 31
getDataForSerialization() (*defcon.Groups method*), 67
getDataForSerialization() (*defcon.Info method*), 57
getDataForSerialization() (*defcon.Kerning method*), 62
getDataForSerialization() (*defcon.Layer method*), 21
getDataForSerialization() (*defcon.LayerSet method*), 93
getDataForSerialization() (*defcon.Lib method*), 75
getDataForSerialization() (*defcon.objects.base.BaseObject method*), 90
getDataForSerialization() (*defcon.UnicodeData method*), 81
getHeldNotificationNotes() (*defcon.tools.notifications.NotificationCenter method*), 86
getHeldNotifications() (*defcon.tools.notifications.NotificationCenter method*), 86
getPen() (*defcon.Glyph method*), 31
getPointPen() (*defcon.Glyph method*), 31
getRepresentation() (*defcon.Anchor method*), 50
getRepresentation() (*defcon.Component method*), 44
getRepresentation() (*defcon.Contour method*), 39
getRepresentation() (*defcon.Features method*), 71
getRepresentation() (*defcon.Font method*), 13
getRepresentation() (*defcon.Glyph method*), 31
getRepresentation() (*defcon.Groups method*), 67
getRepresentation() (*defcon.Info method*), 57
getRepresentation() (*defcon.Kerning method*), 62

getRepresentation() (*defcon.Layer method*), 21
getRepresentation() (*defcon.LayerSet method*), 93
getRepresentation() (*defcon.Lib method*), 75
getRepresentation() (*defcon.objects.base.BaseObject method*), 90
getRepresentation() (*defcon.UnicodeData method*), 81
getSaveProgressBarTickCount() (*defcon.Font method*), 13
getSaveProgressBarTickCount() (*defcon.Layer method*), 21
getSaveProgressBarTickCount() (*defcon.LayerSet method*), 93
Glyph (*class in defcon*), 26
glyph (*defcon.Anchor attribute*), 50
glyph (*defcon.Component attribute*), 44
glyph (*defcon.Contour attribute*), 39
glyph (*defcon.Lib attribute*), 75
glyphNameForForcedUnicode() (*defcon.UnicodeData method*), 81
glyphNameForUnicode() (*defcon.UnicodeData method*), 81
glyphOrder (*defcon.Font attribute*), 13
glyphsWithOutlines (*defcon.Font attribute*), 13
glyphsWithOutlines (*defcon.Layer attribute*), 21
Groups (*class in defcon*), 64
groups (*defcon.Font attribute*), 13
guidelineClass (*defcon.Glyph attribute*), 31
guidelineIndex() (*defcon.Font method*), 14
guidelineIndex() (*defcon.Glyph method*), 31
guidelines (*defcon.Font attribute*), 14
guidelines (*defcon.Glyph attribute*), 31
guidelines (*defcon.Info attribute*), 58

H

hasCachedRepresentation() (*defcon.Anchor method*), 50
hasCachedRepresentation() (*defcon.Component method*), 44
hasCachedRepresentation() (*defcon.Contour method*), 39
hasCachedRepresentation() (*defcon.Features method*), 71
hasCachedRepresentation() (*defcon.Font method*), 14
hasCachedRepresentation() (*defcon.Glyph method*), 31
hasCachedRepresentation() (*defcon.Groups method*), 67
hasCachedRepresentation() (*defcon.Info method*), 58
hasCachedRepresentation() (*defcon.Kerning method*), 62
hasCachedRepresentation() (*defcon.Layer method*), 21
hasCachedRepresentation() (*defcon.LayerSet method*), 93
hasCachedRepresentation() (*defcon.Lib method*), 75
hasCachedRepresentation() (*defcon.objects.base.BaseObject method*), 90
hasCachedRepresentation() (*defcon.UnicodeData method*), 81
hasObserver() (*defcon.Anchor method*), 50
hasObserver() (*defcon.Component method*), 44
hasObserver() (*defcon.Contour method*), 39
hasObserver() (*defcon.Features method*), 71
hasObserver() (*defcon.Font method*), 14
hasObserver() (*defcon.Glyph method*), 31
hasObserver() (*defcon.Groups method*), 67
hasObserver() (*defcon.Info method*), 58
hasObserver() (*defcon.Kerning method*), 62
hasObserver() (*defcon.Layer method*), 21
hasObserver() (*defcon.LayerSet method*), 94
hasObserver() (*defcon.Lib method*), 75
hasObserver() (*defcon.objects.base.BaseObject method*), 90
hasObserver() (*defcon.tools.notifications.NotificationCenter method*), 87
hasObserver() (*defcon.UnicodeData method*), 81
height (*defcon.Glyph attribute*), 31
holdNotifications() (*defcon.Anchor method*), 50
holdNotifications() (*defcon.Component method*), 45
holdNotifications() (*defcon.Contour method*), 39
holdNotifications() (*defcon.Features method*), 71
holdNotifications() (*defcon.Font method*), 14
holdNotifications() (*defcon.Glyph method*), 31
holdNotifications() (*defcon.Groups method*), 67
holdNotifications() (*defcon.Info method*), 58
holdNotifications() (*defcon.Kerning method*), 62
holdNotifications() (*defcon.Layer method*), 22
holdNotifications() (*defcon.LayerSet method*), 94
holdNotifications() (*defcon.Lib method*), 75
holdNotifications() (*defcon.objects.base.BaseObject method*), 90
holdNotifications() (*defcon.tools.notifications.NotificationCenter method*), 87
holdNotifications() (*defcon.UnicodeData method*), 81

I

identifier (*defcon.Anchor attribute*), 51
identifier (*defcon.Component attribute*), 45
identifier (*defcon.Contour attribute*), 39
identifier (*defcon.Point attribute*), 47
identifiers (*defcon.Anchor attribute*), 51
identifiers (*defcon.Component attribute*), 45
identifiers (*defcon.Contour attribute*), 39
identifiers (*defcon.Font attribute*), 14
identifiers (*defcon.Glyph attribute*), 31
image (*defcon.Glyph attribute*), 32
imageClass (*defcon.Glyph attribute*), 32
imageReferences (*defcon.Layer attribute*), 22
images (*defcon.Font attribute*), 14
index() (*defcon.Contour method*), 39
Info (*class in defcon*), 56
info (*defcon.Font attribute*), 14
insertAnchor() (*defcon.Glyph method*), 32
insertComponent() (*defcon.Glyph method*), 32
insertContour() (*defcon.Glyph method*), 32
insertGlyph() (*defcon.Font method*), 14
insertGlyph() (*defcon.Layer method*), 22
insertGuideline() (*defcon.Font method*), 14
insertGuideline() (*defcon.Glyph method*), 32
insertPoint() (*defcon.Contour method*), 39
items() (*defcon.Anchor method*), 51
items() (*defcon.Groups method*), 67
items() (*defcon.Kerning method*), 62
items() (*defcon.Lib method*), 76
items() (*defcon.UnicodeData method*), 81

K

Kerning (*class in defcon*), 59
kerning (*defcon.Font attribute*), 14
kerningGroupConversionRenameMaps (*defcon.Font attribute*), 14
keys() (*defcon.Anchor method*), 51
keys() (*defcon.Groups method*), 67
keys() (*defcon.Kerning method*), 62
keys() (*defcon.Layer method*), 22
keys() (*defcon.Lib method*), 76
keys() (*defcon.UnicodeData method*), 82

L

Layer (*class in defcon*), 19
layer (*defcon.Anchor attribute*), 51
layer (*defcon.Component attribute*), 45
layer (*defcon.Contour attribute*), 39
layer (*defcon.Glyph attribute*), 32
layer (*defcon.Lib attribute*), 76
layer (*defcon.UnicodeData attribute*), 82
layerOrder (*defcon.LayerSet attribute*), 94
layers (*defcon.Font attribute*), 15
LayerSet (*class in defcon*), 91
layerSet (*defcon.Anchor attribute*), 51
layerSet (*defcon.Component attribute*), 45
layerSet (*defcon.Contour attribute*), 39
layerSet (*defcon.Glyph attribute*), 32
layerSet (*defcon.Layer attribute*), 22
layerSet (*defcon.Lib attribute*), 76
layerSet (*defcon.UnicodeData attribute*), 82
leftMargin (*defcon.Glyph attribute*), 32
Lib (*class in defcon*), 73
lib (*defcon.Font attribute*), 15
lib (*defcon.Glyph attribute*), 32
lib (*defcon.Layer attribute*), 22
libClass (*defcon.Glyph attribute*), 32
loadGlyph() (*defcon.Layer method*), 22

M

markColor (*defcon.Glyph attribute*), 32
move() (*defcon.Anchor method*), 51
move() (*defcon.Component method*), 45
move() (*defcon.Contour method*), 40
move() (*defcon.Glyph method*), 32
move() (*defcon.Point method*), 47

N

name (*defcon.Anchor attribute*), 51
name (*defcon.Glyph attribute*), 32
name (*defcon.Layer attribute*), 22
name (*defcon.Point attribute*), 47
name (*defcon.tools.notifications.Notification attribute*), 87
newGlyph() (*defcon.Font method*), 15
newGlyph() (*defcon.Layer method*), 22
newLayer() (*defcon.Font method*), 15
newLayer() (*defcon.LayerSet method*), 94
note (*defcon.Glyph attribute*), 33
Notification (*class in defcon.tools.notifications*), 87
NotificationCenter (*class in defcon.tools.notifications*), 85

O

object (*defcon.tools.notifications.Notification attribute*), 87
onCurvePoints (*defcon.Contour attribute*), 40
open (*defcon.Contour attribute*), 40
openRelativeForGlyphName() (*defcon.UnicodeData method*), 82

P

path (*defcon.Font attribute*), 15
Point (*class in defcon*), 47
pointClass (*defcon.Contour attribute*), 40
pointClass (*defcon.Glyph attribute*), 33

`pointInside()` (*defcon.Component method*), 45
`pointInside()` (*defcon.Contour method*), 40
`pointInside()` (*defcon.Glyph method*), 33
`pop()` (*defcon.Anchor method*), 51
`pop()` (*defcon.Groups method*), 67
`pop()` (*defcon.Kerning method*), 62
`pop()` (*defcon.Lib method*), 76
`pop()` (*defcon.UnicodeData method*), 82
`popitem()` (*defcon.Anchor method*), 51
`popitem()` (*defcon.Groups method*), 67
`popitem()` (*defcon.Kerning method*), 62
`popitem()` (*defcon.Lib method*), 76
`popitem()` (*defcon.UnicodeData method*), 82
`positionForProspectivePointInsertionAtSegmentAndT()` (*defcon.Contour method*), 40
`postNotification()` (*defcon.Anchor method*), 51
`postNotification()` (*defcon.Component method*), 45
`postNotification()` (*defcon.Contour method*), 40
`postNotification()` (*defcon.Features method*), 71
`postNotification()` (*defcon.Font method*), 15
`postNotification()` (*defcon.Glyph method*), 33
`postNotification()` (*defcon.Groups method*), 67
`postNotification()` (*defcon.Info method*), 58
`postNotification()` (*defcon.Kerning method*), 63
`postNotification()` (*defcon.Layer method*), 22
`postNotification()` (*defcon.LayerSet method*), 94
`postNotification()` (*defcon.Lib method*), 76
`postNotification()` (*defcon.objects.base.BaseObject method*), 90
`postNotification()` (*defcon.UnicodeData method*), 82
`pseudoUnicodeForGlyphName()` (*defcon.UnicodeData method*), 82

R

`redo()` (*defcon.Anchor method*), 51
`redo()` (*defcon.Component method*), 45
`redo()` (*defcon.Contour method*), 40
`redo()` (*defcon.Features method*), 72
`redo()` (*defcon.Font method*), 15
`redo()` (*defcon.Glyph method*), 33
`redo()` (*defcon.Groups method*), 68
`redo()` (*defcon.Info method*), 58
`redo()` (*defcon.Kerning method*), 63
`redo()` (*defcon.Layer method*), 23
`redo()` (*defcon.LayerSet method*), 94
`redo()` (*defcon.Lib method*), 76
`redo()` (*defcon.objects.base.BaseObject method*), 90
`redo()` (*defcon.UnicodeData method*), 82
`releaseHeldNotifications()` (*defcon.Anchor method*), 51
`releaseHeldNotifications()` (*defcon.Component method*), 45
`releaseHeldNotifications()` (*defcon.Contour method*), 40
`releaseHeldNotifications()` (*defcon.Features method*), 72
`releaseHeldNotifications()` (*defcon.Font method*), 15
`releaseHeldNotifications()` (*defcon.Glyph method*), 33
`releaseHeldNotifications()` (*defcon.Groups method*), 68
`releaseHeldNotifications()` (*defcon.Info method*), 58
`releaseHeldNotifications()` (*defcon.Kerning method*), 63
`releaseHeldNotifications()` (*defcon.Layer method*), 23
`releaseHeldNotifications()` (*defcon.LayerSet method*), 94
`releaseHeldNotifications()` (*defcon.Lib method*), 76
`releaseHeldNotifications()` (*defcon.objects.base.BaseObject method*), 90
`releaseHeldNotifications()` (*defcon.tools.notifications.NotificationCenter method*), 87
`releaseHeldNotifications()` (*defcon.UnicodeData method*), 82
`reloadData()` (*defcon.Font method*), 15
`reloadFeatures()` (*defcon.Font method*), 15
`reloadGlyphs()` (*defcon.Font method*), 15
`reloadGlyphs()` (*defcon.Layer method*), 23
`reloadGroups()` (*defcon.Font method*), 16
`reloadImages()` (*defcon.Font method*), 16
`reloadInfo()` (*defcon.Font method*), 16
`reloadKerning()` (*defcon.Font method*), 16
`reloadLayers()` (*defcon.Font method*), 16
`reloadLayers()` (*defcon.LayerSet method*), 95
`reloadLib()` (*defcon.Font method*), 16
`removeAnchor()` (*defcon.Glyph method*), 33
`removeComponent()` (*defcon.Glyph method*), 33
`removeContour()` (*defcon.Glyph method*), 33
`removeGlyphData()` (*defcon.UnicodeData method*), 82
`removeGuideline()` (*defcon.Font method*), 16
`removeGuideline()` (*defcon.Glyph method*), 33
`removeObserver()` (*defcon.Anchor method*), 51
`removeObserver()` (*defcon.Component method*), 46
`removeObserver()` (*defcon.Contour method*), 40
`removeObserver()` (*defcon.Features method*), 72
`removeObserver()` (*defcon.Font method*), 16
`removeObserver()` (*defcon.Glyph method*), 33
`removeObserver()` (*defcon.Groups method*), 68
`removeObserver()` (*defcon.Info method*), 58
`removeObserver()` (*defcon.Kerning method*), 63

removeObserver() (*defcon.Layer method*), 23
removeObserver() (*defcon.LayerSet method*), 95
removeObserver() (*defcon.Lib method*), 76
removeObserver() (*defcon.objects.base.BaseObject method*), 91
removeObserver() (*defcon.tools.notifications.NotificationCenter method*), 87
removeObserver() (*defcon.UnicodeData method*), 82
removePoint() (*defcon.Contour method*), 41
removeSegment() (*defcon.Contour method*), 41
representationKeys() (*defcon.Anchor method*), 52
representationKeys() (*defcon.Component method*), 46
representationKeys() (*defcon.Contour method*), 41
representationKeys() (*defcon.Features method*), 72
representationKeys() (*defcon.Font method*), 16
representationKeys() (*defcon.Glyph method*), 34
representationKeys() (*defcon.Groups method*), 68
representationKeys() (*defcon.Info method*), 59
representationKeys() (*defcon.Kerning method*), 63
representationKeys() (*defcon.Layer method*), 23
representationKeys() (*defcon.LayerSet method*), 95
representationKeys() (*defcon.Lib method*), 77
representationKeys() (*defcon.objects.base.BaseObject method*), 91
representationKeys() (*defcon.UnicodeData method*), 83
reverse() (*defcon.Contour method*), 41
rightMargin (*defcon.Glyph attribute*), 34

S

save() (*defcon.Font method*), 16
save() (*defcon.Layer method*), 23
save() (*defcon.LayerSet method*), 95
saveData() (*defcon.Font method*), 17
saveFeatures() (*defcon.Font method*), 17
saveGlyph() (*defcon.Layer method*), 23
saveGroups() (*defcon.Font method*), 17
saveImages() (*defcon.Font method*), 17
saveInfo() (*defcon.Font method*), 17
saveKerning() (*defcon.Font method*), 17
saveLib() (*defcon.Font method*), 17
scriptForGlyphName() (*defcon.UnicodeData method*), 83
segments (*defcon.Contour attribute*), 41
segmentType (*defcon.Point attribute*), 47
setDataFromSerialization() (*defcon.Anchor method*), 52
setDataFromSerialization() (*defcon.Component method*), 46
setDataFromSerialization() (*defcon.Contour method*), 41
setDataFromSerialization() (*defcon.Features method*), 72
setDataFromSerialization() (*defcon.Font method*), 17
setDataFromSerialization() (*defcon.Glyph method*), 34
setDataFromSerialization() (*defcon.Groups method*), 68
setDataFromSerialization() (*defcon.Info method*), 59
setDataFromSerialization() (*defcon.Kerning method*), 63
setDataFromSerialization() (*defcon.Layer method*), 23
setDataFromSerialization() (*defcon.LayerSet method*), 95
setDataFromSerialization() (*defcon.Lib method*), 77
setDataFromSerialization() (*defcon.objects.base.BaseObject method*), 91
setDataFromSerialization() (*defcon.UnicodeData method*), 83
setdefault() (*defcon.Anchor method*), 52
setdefault() (*defcon.Groups method*), 68
setdefault() (*defcon.Kerning method*), 63
setdefault() (*defcon.Lib method*), 77
setdefault() (*defcon.UnicodeData method*), 83
setStartPoint() (*defcon.Contour method*), 41
smooth (*defcon.Point attribute*), 47
sortGlyphNames() (*defcon.UnicodeData method*), 83
splitAndInsertPointAtSegmentAndT() (*defcon.Contour method*), 41

T

tempLib (*defcon.Font attribute*), 17
tempLib (*defcon.Glyph attribute*), 34
tempLib (*defcon.Layer attribute*), 23
testForExternalChanges() (*defcon.Font method*), 17
testForExternalChanges() (*defcon.Layer method*), 23
testForExternalChanges() (*defcon.LayerSet method*), 95
text (*defcon.Features attribute*), 72
topMargin (*defcon.Glyph attribute*), 34
transformation (*defcon.Component attribute*), 46

U

ufoFileStructure (*defcon.Font attribute*), 18
ufoFormatVersion (*defcon.Font attribute*), 18
ufoFormatVersionTuple (*defcon.Font attribute*), 18
undo() (*defcon.Anchor method*), 52
undo() (*defcon.Component method*), 46
undo() (*defcon.Contour method*), 41
undo() (*defcon.Features method*), 72
undo() (*defcon.Font method*), 18
undo() (*defcon.Glyph method*), 34
undo() (*defcon.Groups method*), 68
undo() (*defcon.Info method*), 59
undo() (*defcon.Kerning method*), 63
undo() (*defcon.Layer method*), 23
undo() (*defcon.LayerSet method*), 95
undo() (*defcon.Lib method*), 77
undo() (*defcon.objects.base.BaseObject method*), 91
undo() (*defcon.UnicodeData method*), 85
undoManager (*defcon.Anchor attribute*), 52
undoManager (*defcon.Component attribute*), 46
undoManager (*defcon.Contour attribute*), 41
undoManager (*defcon.Features attribute*), 72
undoManager (*defcon.Font attribute*), 18
undoManager (*defcon.Glyph attribute*), 34
undoManager (*defcon.Groups attribute*), 68
undoManager (*defcon.Info attribute*), 59
undoManager (*defcon.Kerning attribute*), 63
undoManager (*defcon.Layer attribute*), 23
undoManager (*defcon.LayerSet attribute*), 95
undoManager (*defcon.Lib attribute*), 77
undoManager (*defcon.objects.base.BaseObject attribute*), 91
undoManager (*defcon.UnicodeData attribute*), 85
unicode (*defcon.Glyph attribute*), 34
UnicodeData (*class in defcon*), 78
unicodeData (*defcon.Font attribute*), 18
unicodeData (*defcon.Layer attribute*), 23
unicodeForGlyphName() (*defcon.UnicodeData method*), 85
unicodes (*defcon.Glyph attribute*), 34
update() (*defcon.Anchor method*), 52
update() (*defcon.Groups method*), 68
update() (*defcon.Kerning method*), 63
update() (*defcon.Lib method*), 77
update() (*defcon.UnicodeData method*), 85
updateGlyphOrder() (*defcon.Font method*), 18

V

values() (*defcon.Anchor method*), 52
values() (*defcon.Groups method*), 68
values() (*defcon.Kerning method*), 64
values() (*defcon.Lib method*), 77
values() (*defcon.UnicodeData method*), 85
verticalOrigin (*defcon.Glyph attribute*), 34

W

width (*defcon.Glyph attribute*), 34

X

x (*defcon.Anchor attribute*), 52
x (*defcon.Point attribute*), 47

Y

y (*defcon.Anchor attribute*), 52
y (*defcon.Point attribute*), 47